AD-A077 224 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 13/9
A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MAN--ETC(U)
DEC 78 C C GESCHKE DAAB07-72-C-0259
UNCLASSIFIED R-837 NL

1 OF 2
AD
A077224

REPORT R-837 DECEMBER, 1978 UILU-ENG 78-2230

CSL COORDINATED SCIENCE LABORATORY

A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MANIPULATORS

CLIFFORD CALVIN GESCHKE

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MANIPULATORS		5. TYPE OF REPORT & PERIOD COVERED 9 Technical Report
14		6. PERFORMING ORG. REPORT NUMBER R-837, UILU-ENG-78-2230
7. AUTHOR(s) 10 Clifford Calvin Geschke	15	8. CONTRACT OR GRANT NUMBER(s) DAAB-07-72-C-0259, DAAG-29-78-C-0016
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory ✓ University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program	12 111	12. REPORT DATE 11 December 1978
		13. NUMBER OF PAGES 102
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Sensor-Based Robot Manipulators
Robot Servo System (RSS)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Although a great deal of effort has been spent studying how various forms of sensory feedback can be used to control a robot manipulator, there exists no unified method for specifying robot tasks in terms of sensory information.

What all past programming schemes have failed to realize is that external sensors fall into the same class as built-in position or velocity sensors. They all provide information about the state of the manipulator and are potentially useful in controlling its activity.

This report presents RSS, a Robot Servo System which is based upon the → over

20. ABSTRACT (continued)

following concepts: 1) The programmer directly specifies sampled-data servos which establish links between sensory information and manipulator action. 2) Servos for controlling the two spatial aspects (position and orientation) and the two kinesthetic aspects (force and torque), may be specified independently. Any conflicts between the various servos are automatically resolved. 3) Any of the servos may use whatever sensory data is desired. 4) Manipulator positions are specified as the coincidence of some reference point with a goal. Orientation is specified as the alignment of vectors.

Using this system, a programmer may specify powerful sensor-driven robot tasks in a manner which corresponds closely to his intuitive ideas about how the task should be performed.

UILU-ENG 78-2230

A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MANIPULATORS

by

Clifford Calvin Geschke

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and Contract DAAG-29-78-C-0016.

A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MANIPULATORS

BY

CLIFFORD CALVIN GESCHKE

B.S., Purdue University, 1973
M.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1979

Thesis Adviser: Professor R. T. Chien

Urbana, Illinois

A SYSTEM FOR PROGRAMMING AND CONTROLLING SENSOR-BASED ROBOT MANIPULATORS

Clifford Calvin Geschke, Ph.D.
Coordinated Science Laboratory and Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1979

Although a great deal of effort has been spent studying how various forms of sensory feedback can be used to control a robot manipulator, there exists no unified method for specifying robot tasks in terms of sensory information.

What all past programming schemes have failed to realize is that external sensors fall into the same class as built-in position or velocity sensors. They all provide information about the state of the manipulator and are potentially useful in controlling its activity.

This thesis presents RSS, a Robot Servo System which is based upon the following concepts: 1) The programmer directly specifies sampled-data servos which establish links between sensory information and manipulator action. 2) Servos for controlling the two spatial aspects (position and orientation) and the two kinesthetic aspects (force and torque), may be specified independently. Any conflicts between the various servos are automatically resolved. 3) Any of the servos may use whatever sensory data is desired. 4) Manipulator positions are specified as the coincidence of some reference point with a goal. Orientation is specified as the alignment of vectors.

Using this system, a programmer may specify powerful sensor-driven robot tasks in a manner which corresponds closely to his intuitive ideas about how the task should be performed.

ACKNOWLEDGMENT

I would like to thank my advisor Dr. R. T. Chien for providing the support for my thesis research, for guiding me during difficult times, and for teaching me much about the realities of life.

I would also like to thank the many people who have helped me during the past several years: Bob Bales, who built the robot and kept it running; Wes Snyder and Don Williamson who helped design the first robot controller; Bill Brew and Rod Fletcher who helped with the new robot controller; and Chuck Jacobus who wrote the operating system, advised me about vision, and provided constant moral support.

Other people whose help deserves mention include: B. Champagne, S. P. Fund, R. T. Gladin, V. C. Jones, B. C. Kuo, J. St. Clair, M. Selander, R. Schmidt, and D. Waltz.

I am grateful to both my parents for their unfailing confidence in me, and to my father for his help with the illustrations in this thesis.

Finally I wish to thank my wife, Sue. Without her love and understanding, I never would have finished.

TABLE OF CONTENTS

Page

1. INTRODUCTION

Many research groups have studied how feedback from external sensors can be used to improve the performance of robot manipulators; however, relatively few groups have studied how to use these sensors when programming the robot. Most programming methods do not include external sensors as part of the basic language definition, but instead rely on special subroutines or messages from external programs.

Industrial robot systems are typically programmed by recording a sequence of motions for later playback. These systems can use external sensors only as part of special functions such as 'move until touch' or 'wait for signal'.

Even the most advanced robot programming languages such as AL and its predecessor WAVE, are designed primarily to deal with manipulator positions and velocities. In both AL and WAVE, force control or compliance is achieved by overriding portions of the position servo. Neither system permits visual data to be used directly, but requires an external program to change internal variables in the manipulator program.

What all past programming schemes have failed to realize is that external sensors fall into the same class as built-in position or velocity sensors. They all provide information about the state of the manipulator with respect to its environment, and they all are potentially useful in controlling the manipulator activity. If data from the vision system or force sensors is treated the same as data from the built-in sensors, there is nothing to prevent programming a trajectory using vision or force feedback.

Another drawback of existing methods is their failure to deal with the distinction between the spatial and kinesthetic aspects of manipulation [1]. The spatial aspect deals with parts which are not in contact. The goal is to establish some relation between the positions and orientations of the parts. The kinesthetic aspect deals with parts which are in contact. The goal is to establish certain forces or torques between the parts. Real tasks often combine these two aspects. For example, in a drilling operation, it is important to maintain the proper orientation of the drill and to keep it positioned at the hole, but it is also necessary to apply a force and to move along the hole axis. It should be obvious that vision is often appropriate for determining spatial relations, and force sensors are appropriate for determining kinesthetic relations. A programming method should allow the different aspects of manipulation to be controlled by whatever sensor is appropriate at the time.

Within the foreseeable future, robot programs will be written by human programmers; therefore, it is important that the programming method appeal to the user's intuition. Manipulator commands should correspond closely to actual assembly or manufacturing operations, and data from the sensors should be presented in an understandable manner. The programming method should be independent of the particular manipulator geometry or sensors, but should still allow a knowledgeable programmer to make use of the robot's special features.

If high level manufacturing or assembly programs are developed, it will be necessary for them to specify manipulator action in terms of sensory data. An intermediate language, which allows such specification in a configuration independent manner, would allow the same high level system to generate programs for different manipulators. Also, if humans could understand the programs generated, the high level system would be much easier to debug and maintain.

Briefly restated, a programming method for a sensor-based robot manipulator system should have the following properties:

1. Information from all sensors, both external and built-in, should be treated similarly. All information should be available to control the manipulator.

2. It should be possible to control both the spatial and kinesthetic aspects of manipulation, and combine them as necessary.

3. It should be possible to control any aspect of the manipulator using whatever sensor is appropriate at the time.

4. Programs should be easily understood by humans.

This thesis presents RSS, a Robot Servo System for programming and controlling a sensor-based robot. RSS exhibits the properties listed above because it is based upon the following new concepts in robot manipulation:

1. The programmer directly specifies sampled-data servos which establish a link between sensory information and manipulator action.

2. Servos for controlling the two spatial aspects (position and orientation) and the two kinesthetic aspects (force and torque), may all be specified independently. Any conflicts between the various servos are automatically resolved.

3. Any of the servos may use whatever sensory data is desired.

4. Manipulator positions are specified as the coincidence of some reference point with a goal. Orientation is specified as the alignment of vectors.

In RSS, a robot program consists of declarations to establish servos, and wait statements to suspend execution until some condition is satisfied. Usually the condition is that an error term in the servo has been made sufficiently small. The programmer may also establish condition monitors which asynchronously alter normal program flow when a specified condition is satisfied.

To facilitate interaction with human programmers, and to provide independence from a particular manipulator and sensor geometry, all positions and directions in RSS are specified in terms of a standard three dimensional Cartesian coordinate system. All sensor data, servo declarations, and condition monitors deal with three-element vectors or scalars.

No attempt has been made to embed RSS within a block structured language such as PL/1 or ALGOL, or to provide the coordinate transformation capabilities of AL. In fact, relatively little effort has been put into designing the actual language syntax. While such additions may be desired in a production system, they are not the real issues in using sensory feedback.

The RSS described in this thesis is not mere speculation or intention. It is an actual working system running on a PDP11/40 minicomputer in conjuction with a Stanford electric arm [2]. An outline of the arm is shown in Figure 1. In addition to the built-in position and velocity sensors, the system has access to force and touch information, and data from a real-time vision system which runs on a PDP10 computer.

By expressing manipulator control in terms of arbitrary sensory feedback, in a manner which is natural for manipulator tasks, RSS demonstrates that its fundamental concepts will be useful in designing future robot programming languages.

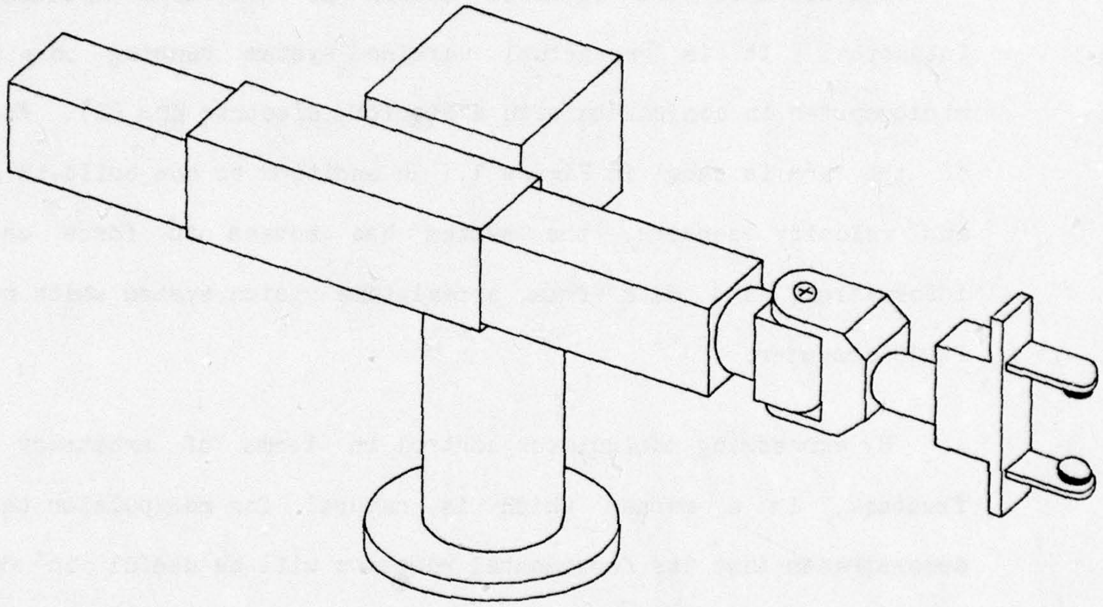

Figure 1 - Manipulator Outline

2. STATE-OF-THE-ART

Robot manipulators are becoming increasingly popular in manufacturing applications. Over the past decade, the cost of powerful computers has fallen sharply, while the cost of conventional manufacturing techniques has continued to rise. Studies have shown that a significant number of manufacturing tasks could be performed more economically with programmable robot systems than with either manual labor or full scale automatic machinery [3,4,5].

The key to a successful robot system is generality. Full scale specialized automation may be the most economical when manufacturing large numbers of identical items, but the time and expense of modifying such automation precludes using the same equipment for smaller numbers of items. In fact, up to 75 percent of all manufacturing is done in small batches [6]. When frequent product changes are required, computer controlled programmable automation is often the solution. If this automation takes the form of a general purpose robot system, the benefits of interchangeability and modularity are also realized.

2.1 SENSORS IN ROBOTICS

Most of the robot systems in industry today are little more than extensions of numerically controlled machine technology. They blindly perform predetermined actions, and depend upon precise control of both the robot's position, and the position of the objects with which it interacts. Slight misalignments usually result in catastrophic errors. While such robots are useful, they are limited in their applications.

In order to avoid the cost of precisely controlling the manufacturing environment, and to provide the capability of dealing with parts variation, feedback from external sensors is used. These sensors include vision, force, touch, and proximity sensors, which provide information about the relationship of a robot manipulator to its environment [7]. Even when sensors are available and a good method for using them has been determined, it is still necessary to write programs for individual manipulator tasks.

2.2 FORCE FEEDBACK

Force feedback for manipulator control has been studied by many research groups [8,9,10,11]. The first manipulator language to provide a sophisticated method of force control was WAVE [12,13]. Although it is primarily concerned with position control and trajectory calculation, WAVE also allows force control. Certain commands remove joints from the position servo and permit translational or rotational compliance. These commands require vector arguments which specify the direction or axis of compliance. Using the planned manipulator position, the joints which are most sensitive to the compliant motions are identified and removed from the position servo. Additionally, a force or torque can be applied with these compliant joints. In some newer work, Paul and Shimano have proposed a method of dynamically compensating for coupling between compliant joints and non-compliant joints [14]. WAVE also has the ability to stop a motion if the force or torque along an axis exceeds a specified value.

Although it features a vastly improved syntax and control structure, Stanford's latest manipulator language, AL, offers no more capability than WAVE in the area of force control [15]. AL does offer greater flexibility in declaring condition monitors which allow arbitrary manipulator action if a monitored force or torque condition is satisfied.

2.3 VISUAL FEEDBACK

Computer vision is also being investigated for feedback control of robot manipulators. Since vision is potentially quite accurate, and since it can be used without disturbing the workspace, it promises to be of great value in controlling the spatial aspects of manipulation. Unfortunately, image processing is so time-consuming that almost all examples of visual feedback have been of the stop-and-look variety [16,17,18]. The robot moves into view and stops, waiting for a picture to be taken and processed. After several seconds, the vision system sends some correction signal to the robot which makes the correction and stops again. This procedure is repeated until the error detected by the vision system is reduced to a sufficiently small amount.

Some notable exceptions to the stop-and-look method are the tracking work of Vince Jones, and some of the latest work at SRI. Jones proposed separating the time consuming task of scene analysis and object recognition from the much faster low level task of finding features such as regions, edges, and corners. He demonstrated that these low level features are sufficient to dynamically control a robot manipulator during stacking and insertion tasks [19]. The latest work at SRI has led to a system which is capable of picking objects off of an uncalibrated moving conveyor belt, using vision [20].

Part of Bolles' work on verification vision has shown how tracking methods can be refined. Given a model of the object, and an estimated position of that object, his system is able to identify which features should be examined and which operators should be applied, in order to refine the estimate to a certain accuracy [21].

All of these visual feedback tasks have been programmed as special cases. Manipulator languages provide no way of dealing directly with visual data. AL and WAVE have both been used in visual feedback tasks, but only by allowing an external program to modify internal variables. In his analysis of AL, Finkel points out the desirability of some sort of general adaptive control scheme. He suggests that external sources might be allowed to dynamically specify a 'nudge' or perturbation from the programmed trajectory [22]. Such a scheme is only useful so long as the sensor is refining a good estimate of the positions indicated at programming time. If the sensor is to provide the gross as well as fine positioning information, some other method is needed.

2.4 HIGH LEVEL LANGUAGES

The very high level languages developed at Stanford [23], MIT [24], and IBM [25], use geometric descriptions of the objects in the workspace and knowledge about the manipulator capabilities to plan the details of manipulator motion. Generally, they generate code for a lower level language such as AL or MAPLE. Unless a low level language has the ability to deal with sensory feedback, the high level languages will be unable to effectively deal with uncertainties in their environment.

3. SOME EXAMPLES

In order to understand how tasks are performed using RSS, several examples will be discussed. The first is a hypothetical circuit card insertion task, simplified to illustrate the fundamental concepts. The remaining examples are reproductions of working RSS programs.

3.1 CIRCUIT CARD INSERTION BY HUMANS

Before discussing circuit card insertion by a robot manipulator, consider how a human might perform the task. After briefly looking at the card in his hand, and at the edge connector, he rotates his hand so that the card is oriented properly for insertion, and he moves the card to a position in front of the connector slot. All of this action takes place without conscious thought. The human knows that to insert the card, he must align its front edge with the slot and its side edge with the insertion axis, and he knows how to move his hand to accomplish this alignment. He then pushes the card into the slot, allowing it to comply to side forces as it slides in. When he feels the card firmly seated in the slot, he knows the task is complete. If extreme precision were required, he could constantly watch the card during insertion and correct any errors.

3.2 SIMPLIFIED CIRCUIT CARD INSERTION

Now consider how a robot manipulator could be programmed to perform the same task. In this example, possible error conditions are ignored, and the complicated motions required to actually insert a tight-fitting circuit card are eliminated so that some fundamental RSS concepts can be better illustrated.

Initially assume that the robot has no vision system and that the position of the card in the hand and the connector are known. Specifically, the end of the edge connector slot is located at robot coordinates (-20,30,5), the slot is aligned with the Z axis (0,0,1), and the insertion direction is aligned with the X axis (1,0,0) as shown in Figure 2. This information may be specified in the following RSS definitions:

```
define SLOTEND = [-20,30,5]
define SLOTAXIS = [0,0,1]
define INSERT = [1,0,0]
```

The circuit card is grasped so that the side edge is aligned with the finger direction and the leading edge is normal to the side of the hand. The card corner to be inserted at SLOTEND is 6 centimeters ahead of the of the finger grasping point and 4 centimeters along the leading edge. This information is written as:

```
define SIDEEDGE = R$FINGER
define LEADEDGE = R$FINGER # R$THUMB
define CORNER = R$GRIP + 6*SIDEEDGE - 4*LEADEDGE
```

As shown in Figure 3, R$FINGER is a vector which points in the finger direction, R$THUMB is a vector which points from finger to finger, and R$GRIP is the location of the robot gripper. R$FINGER # R$THUMB is the vector cross product of the two direction vectors and is normal to the side of the hand, It is also useful to define a point 4 centimeters in front of the slot, along the insertion axis:

```
define FRONT = SLOTEND - 4*INSERT
```

To achieve the proper orientation, a servo is declared to align the leading edge of the circuit card with the slot, and the side edge of the card with the insertion axis:

```
orient fixed LEADEDGE;SIDEEDGE = SLOTAXIS;INSERT
```

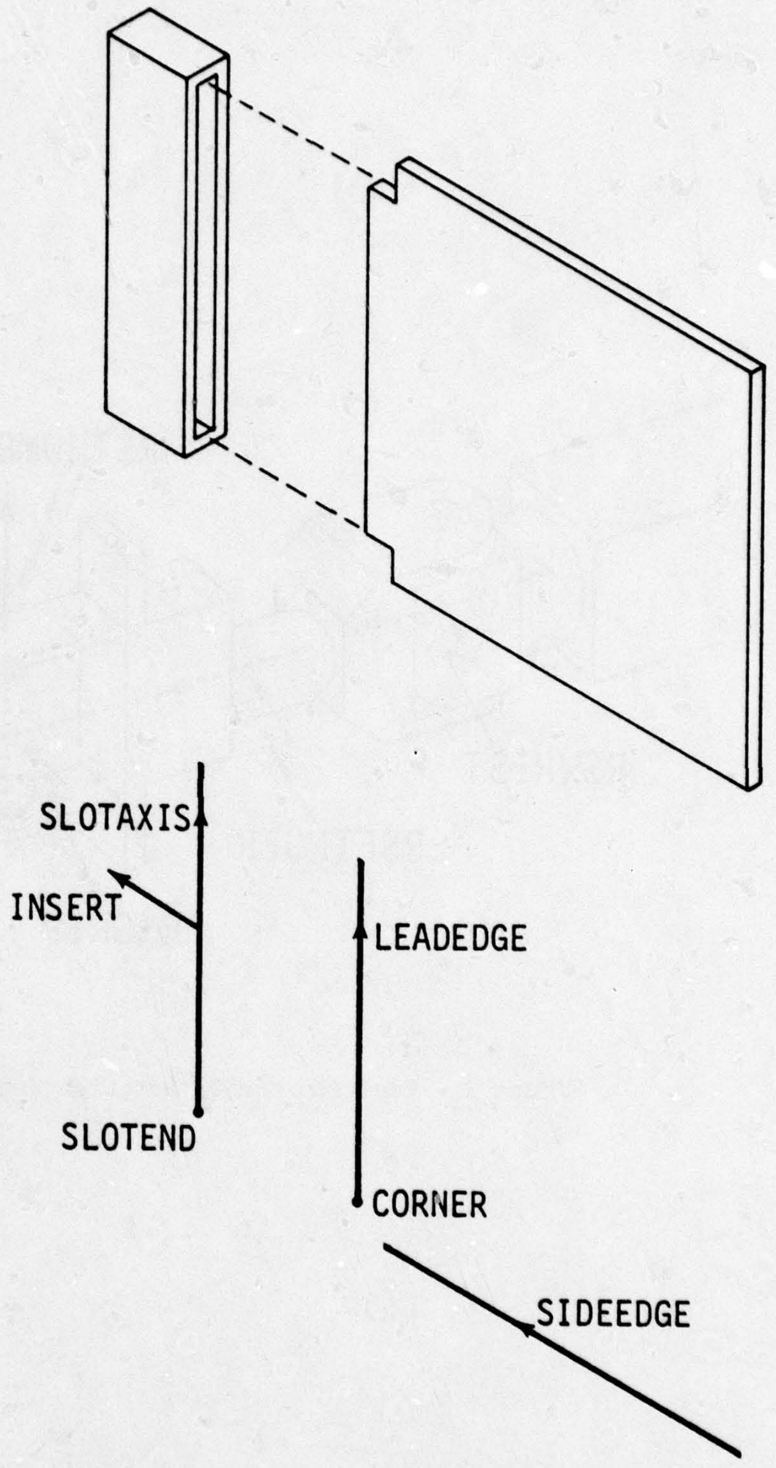

Figure 2 - Circuit Board Insertion Task

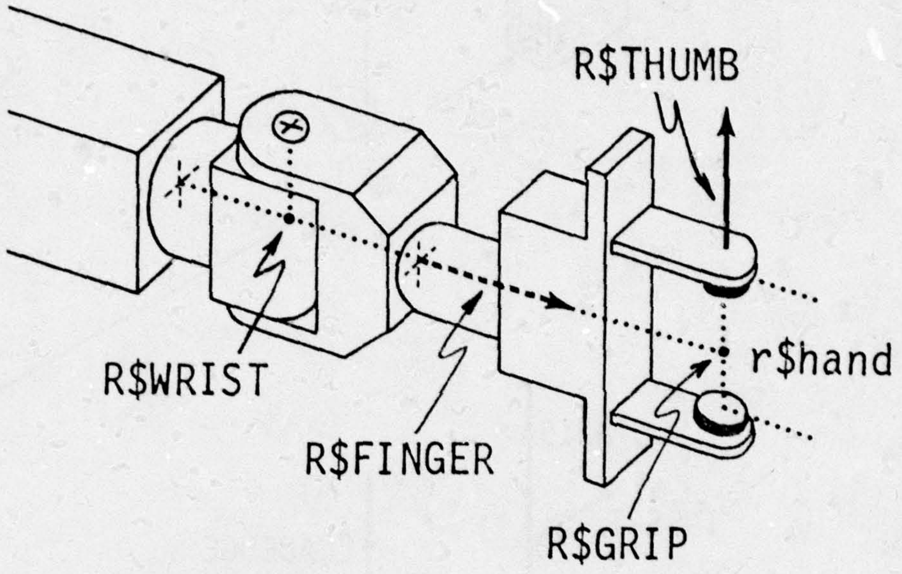

Figure 3 - Position and Orientation Functions

A position servo is declared to bring the card in front of the slot:

```
        position point CORNER = FRONT
```

Once this position is reached, the card is moved toward the slot until contact is made. This motion is accomplished by constraining the corner to a line which passes through the slot end and is parallel to the insertion axis. A force of 20 newtons is applied toward the slot.

```
        position line CORNER = SLOTEND;INSERT
        force 20*INSERT
```

When this force is established, it means that the card is touching the slot. All position constraints are now removed so that the card may comply to forces normal to the insertion direction. Also, the force is increased to 70 newtons to perform the actual insertion. During this time, the previously declared orientation servo is still active.

```
        position none
        force 70*INSERT
```

Once the robot feels an external force equal and opposite to the one it is trying to apply, the task is complete. Figure 4 shows the entire RSS program. The wait statements shown suspend program execution until the relation specified is true. Notice that the accuracy requirement for motion termination may be varied by changing the wait statement relation.

For the sake of comparison, an AL program to perform the same simplified task is shown in Figure 5. The manipulator hand frame is called 'robot' and its coordinate system is defined such that: Z is the direction that the fingers point, Y is the vector from one finger to the other, X is normal to the side of the hand in the direction of the leading edge of the card.

```
define SLOTEND = [-20,30,5]
define SLOTAXIS = [0,0,1]
define INSERT = [1,0,0]
define SIDEEDGE = R$FINGER
define LEADEDGE = R$FINGER # R$THUMB
define CORNER = R$GRIP + 6*SIDEEDGE - 4*LEADEDGE
define FRONT = SLOTEND - 4*INSERT

orient fixed LEADEDGE;SIDEEDGE = SLOTAXIS;INSERT
wait until LEADEDGE.SLOTAXIS; gtr .999;
wait until SIDEEDGE.INSERT; gtr .999;

position point CORNER = FRONT
wait until |FRONT - CORNER|; lss .5;

position line CORNER = SLOTEND;INSERT
force 20*INSERT
wait until R$FORCE.INSERT; lss -20;

position none
force 70*INSERT
wait until R$FORCE.INSERT; lss -50;
```

Figure 4 - Simplified RSS Program for Circuit Card Insertion
Using No Vision

```
FRAME slotend,corner,front;
VECTOR insert;

insert ← VECTOR(1,0,0);
slotend ← FRAME(ROT(Y,-90*DEG)*ROT(X,-90*DEG),
                VECTOR(-20*CM,30*CM,5*CM))
corner ← ROBOT*FRAME(NILROT,VECTOR(0,-4*CM,6*CM))
AFFIX corner TO robot;
front ← slotend - 4*CM*insert;

MOVE corner TO front DIRECTLY;

MOVE corner TO slotend + 4*CM*insert DIRECTLY
        ON FORCE insert > 20*NEWTONS DO STOP;

MOVE corner TO slotend + 4*CM*insert DIRECTLY
        WITH FORCE = 0 ALONG X,Y
        ON FORCE insert > 50*NEWTONS DO STOP;
```

Figure 5 - Simplified AL Program for Circuit Board Insertion Using No Vision

While the AL program is more concise, it sacrifices the ability to vary the termination conditions for each motion. Also, the orientation is specified as a rotation operator rather than the more easily understood alignment of vectors.

The difference between AL and RSS becomes most pronounced when vision is used as feedback. In AL, there is no way to specify that a position or orientation depends upon external data. In RSS, it is simple, and the actual program changes very little. Figure 6 shows a hypothetical RSS program using visual feedback to control position and part of the orientation.

The vision statement simply declares to RSS that the value for that function will be supplied by the vision processor. The locate statement directs the vision processor to scan the entire image looking for the position or orientation of the feature named. The vision processor must already contain information which relates the name to a procedure for identifying the feature. The track command causes the vision processor to continually update the feature value based upon the stream of images being processed. The forget command cancels the previous track command.

The fact that RSS expresses orientation as the alignment of vectors, and position as the coincidence of a point with a goal, greatly simplifies the visual tracking task. Low level features can be identified quickly and used directly by the manipulator servo.

In a more advanced system, which contains models for the objects in the workspace, the high level program could select a procedure for visually locating and tracking objects of interest. After sending this

```
vision SLOTEND, SLOTAXIS, LEADEDGE, SIDEEDGE, CORNER
define INSERT = [1,0,0]
define FRONT = SLOTEND - 4*INSERT

locate SLOTEND
locate SLOTAXIS
locate LEADEDGE
track LEADEDGE
locate SIDEEDGE
track SIDEEDGE
locate CORNER
track CORNER

orient fixed LEADEDGE;SIDEEDGE = SLOTAXIS;INSERT
wait until LEADEDGE.SLOTAXIS; gtr .999;
wait until SIDEEDGE.INSERT; gtr .999;

position point CORNER = FRONT
wait until |FRONT - CORNER|; lss .5;

position line CORNER = SLOTEND;INSERT
force 20*INSERT
wait until R$FORCE.INSERT; lss -20;

position none
forget CORNER
force 70*INSERT
wait until R$FORCE.INSERT; lss -50;
```

Figure 6 - Simplified RSS Program for Circuit Board Insertion Using Vision

procedure to the vision system, an RSS-like system could be used to establish a link between the visual data and the desired manipulator action. In essence, RSS would act like the unconscious portion of a human's mind if he were to perform the task, freeing the conscious portion for planning the overall task.

3.3 ACTUAL CIRCUIT CARD INSERTION

A program which actually performs circuit card insertion is shown in Figure 7. This program uses no vision, but makes extensive use of position constraints and force control. Figure 8 shows the robot performing the task.

The circuit card is assumed to already be in the robot hand when the program begins. Also, the connector slot position 'SLOTEND' must have been explicitly defined before the program begins. The program will handle errors of up to .5 centimeter between the specified slot position and its actual position.

First, the orientation of the card is specified by defining the direction of the side edge of the card (2), the leading edge of the card (3), and the normal to the card top (4), in terms of the robot hand orientation functions shown in Figure 3. The position of the card corner, to be inserted at the slot end, is defined in terms of the robot hand position and the card orientation (5). The orientation of the connector slot is specified by defining the slot axis (6), insertion direction (7), and normal to the slot (8), in terms of the robot coordinate system. These directions are shown in Figure 2. Notice that the orientations are specified in terms of vectors which correspond to

```
; Circuit board insertion program
; Clifford C. Geschke     11-30-78
        define in=2.54                                                  (1)
        define SIDEEDGE=R$FINGER                                        (2)
        define LEADEDGE=R$FINGER#R$THUMB                                (3)
        define TOP=R$THUMB                                              (4)
        define CORNER=R$GRIP+{6.75*in}*SIDEEDGE+{2.5*in}*LEADEDGE       (5)
        define SLOTAXIS=B$Y                                             (6)
        define INSERT=-B$Z                                              (7)
        define NORMAL=SLOTAXIS#INSERT                                   (8)
        print SLOTEND = ;                                               (9)
        type SLOTEND                                                    (10)
        define count=1                                                  (11)
        define down=0                                                   (12)
        define slide=0                                                  (13)
        define wiggle=0                                                 (14)
; set up the force
        force down*INSERT+slide*NORMAL+wiggle*SLOTAXIS                  (15)
        torque ZERO                                                     (16)
; now do the orientation
        orient fixed LEADEDGE;SIDEEDGE=SLOTAXIS;INSERT                  (17)
        servo on                                                        (18)
        wait until SIDEEDGE.INSERT; gtr .999;                           (19)
        wait until LEADEDGE.SLOTAXIS; gtr .999;                         (20)
        print Orientation complete                                      (21)
; now do the positioning
        define FRONT=SLOTEND+{-1*INSERT-.5*SLOTAXIS}                    (22)
        position point CORNER=FRONT                                     (23)
        wait until |CORNER-FRONT|; lss .2;                              (24)
        print Positioned in front of slot                               (25)
; now move until touching slot
        position line CORNER=FRONT;INSERT                               (26)
        define down=30                                                  (27)
        wait until R$FORCE.INSERT; lss -.8*down;                        (28)
        print Touching connector                                        (29)
; move down to adjust orientation
        orient align TOP=NORMAL                                         (30)
        position plane CORNER=FRONT;NORMAL                              (31)
        define down=70                                                  (32)
        define wiggle=-30*R$TORQUE.NORMAL                               (33)
        wait until R$FORCE.INSERT; lss -.8*down;                        (34)
        wait until |R$TORQUE.NORMAL|; lss .1;                           (35)
        orient fixed LEADEDGE;SIDEEDGE=SLOTAXIS;{SIDEEDGE}              (36)
        print Orientation adjusted                                      (37)
```

Figure 7 - Actual Circuit Board Insertion Program

```
; move up slightly and move sideways until we hit the edge
         define FRONT={CORNER-.3*INSERT}
         position line CORNER=FRONT;SLOTAXIS
         wait until (CORNER-FRONT).INSERT; lss .1;
         define wiggle=25
         wait until R$FORCE.SLOTAXIS; lss -.8*wiggle;
         print Touching edge of slot
         define wiggle=-30*R$TORQUE.NORMAL
         wait until |R$TORQUE.NORMAL|; lss .1;
; search sideways if not in slot
         define down=0
         define wiggle=0
         define CENTER={CORNER}
         trap 2 to inpart on (SLOTEND-CORNER).INSERT; lss -.5;
         define inc=0
         position line CORNER=CENTER+inc*NORMAL;INSERT
slide:   define down=20
         wait until R$FORCE.INSERT; lss -.8*down;
         define inc=count*.1
         wait until |(CORNER-CENTER).NORMAL|; lss .25;
         define down=-8
         wait until R$FORCE.INSERT; gtr 0;
         define down=20
         wait until R$FORCE.INSERT; lss -.8*down;
         define inc=-count*.1
         wait until |(CORNER-CENTER).NORMAL|; lss .25;
         define down=-8
         wait until R$FORCE.INSERT; gtr 0;
         print Sliding...
         define count={count+1}
         if count; leq 5; goto slide
         print Cannot find slot... giving up.
         servo off
         stop
; now push down hard
inpart:
         print In part way...
         define count=0
         position none
         define down=70
         define wiggle=0
; make sure hand is straight
         define slide=30*R$TORQUE.SLOTAXIS
         wait until |R$TORQUE.SLOTAXIS|; lss .1
         print Hand is straight
```

Figure 7 Continued

```
inloop: define wiggle=40                                              (78)
        wait until R$FORCE.SLOTAXIS; lss -wiggle/2;                   (79)
        define wiggle=-40                                             (80)
        wait until R$FORCE.SLOTAXIS; gtr wiggle/2;                    (81)
        print Wiggling card...                                        (82)
        if (SLOTEND-CORNER).INSERT; gtr -1.2; goto inloop             (83)
; success
        define count={count+1}                                        (84)
        if count; lss 3; goto inloop                                  (85)
        force 30*R$TORQUE#R$FINGER                                    (86)
        wait until |R$TORQUE|; lss .5                                 (87)
        hand position 2                                               (88)
        wait until r$hand; gtr 1;                                     (89)
        position point R$GRIP={R$GRIP-5*INSERT}                       (90)
        print ALL DONE!                                               (91)
```

Figure 7 continued

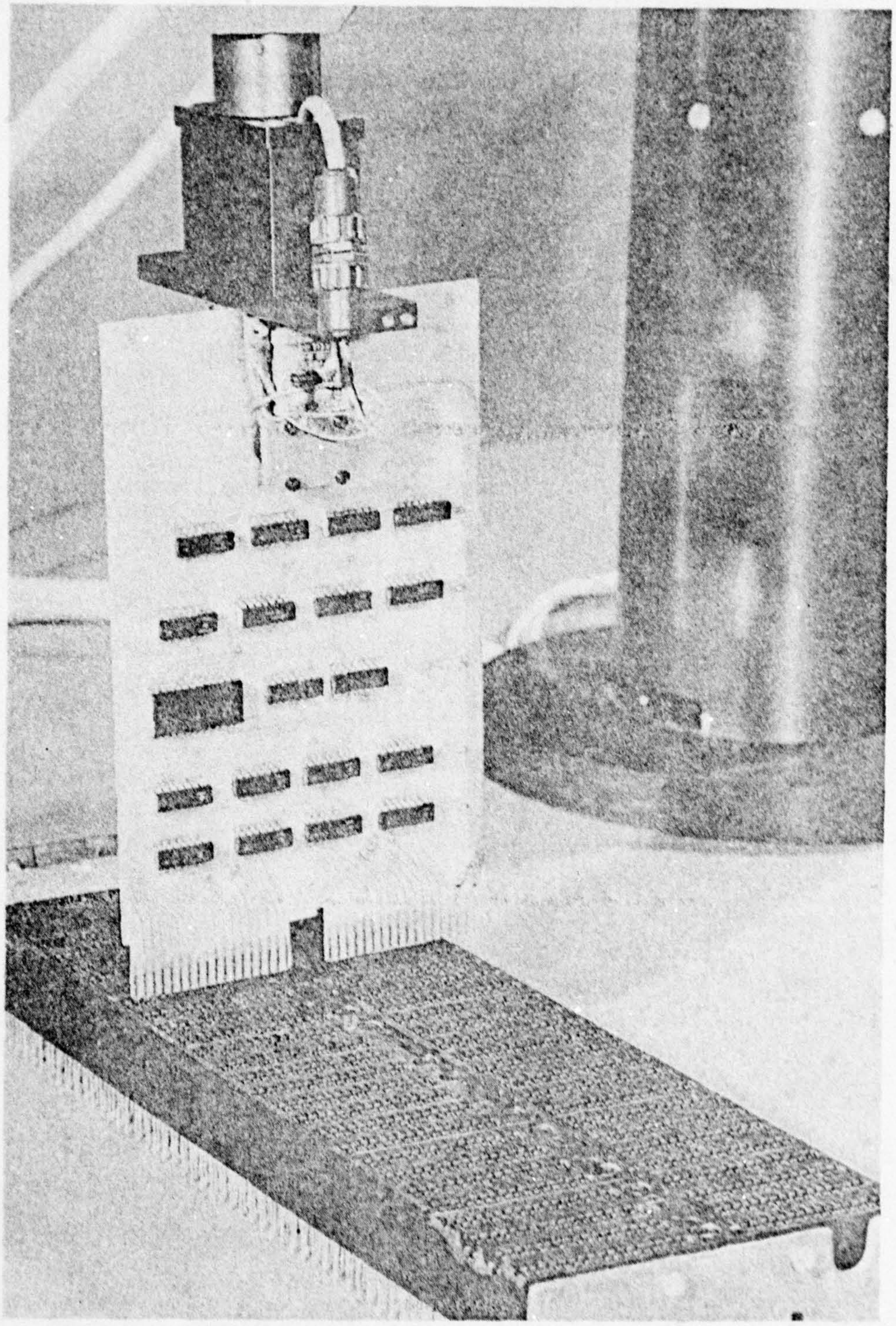

Figure 8 - Photograph of Circuit Board Insertion

features of the objects being manipulated. This method appeals to a programmer's intuition far more than the rotation matrices used by AL.

During the task, is it useful to handle the force as three orthogonal components. The function 'down' is the force along the insertion axis, 'wiggle' is the force along the slot axis, and 'slide' is the force normal to the slot (15). Initially, these three components are set to zero (12-14), but any re-definition of these functions will modify the force servo, so long as force declaration of statement 15 is still in effect.

The first robot action is to rotate the hand so that the leading edge and side edge of the card are aligned with the slot axis and insertion axis of the connector, respectively (17). Next, the hand is moved until the card corner is one centimeter in front of the slot, shifted by .5 centimeter along the slot axis (22-24). This shift insures that the card will not hit the top of the ridge at the edge of the connector. The wait statement (25) suspends program execution until the magnitude of the vector from the card corner to the desired position is less that .2 centimeters. Wait statements allow the programmer great flexibility in determining the completion conditions for each program step.

Now the corner is constrained to the line which passes through this front point and is parallel to the insertion axis (26). Note that the position along the line is unconstrained; therefore, it is possible to apply a force parallel to the line. In RSS, the kinesthetic aspects of manipulation are considered secondary to the spatial aspects. Forces and torques may only be applied within unconstrained degrees of freedom.

This method differs from that of AL and WAVE which apply forces and torques by overriding portions of a completely constrained position and orientation servo. Unlike AL and WAVE, RSS cannot establish inconsistent servos which cause unpredictable motion, such as: move in the X direction while applying a force in the -X direction.

While the position is constrained to the line, a force of 30 newtons is applied along the insertion axis (27) and the program waits until an opposing force of similar magnitude is detected (28). At this point, the card is touching the connector.

The next step is to correct the orientation of the card by allowing it to rotate about its normal axis (30), pushing down with a force of 70 newtons (32), and applying a force along the slot axis which will null out any torque about the normal axis (33) which results from pushing down. To establish all these forces, it is necessary to relax the position constraint and allow the card corner to move within a plane which passes through the front point and is normal to the slot normal (31).

Note that statement 33 establishes a link between the wrist torque sensor and the wrist force servo. The ability to establish links between arbitrary sensors and any aspect of the manipulator is one of the most important properties of RSS.

After waiting until the proper forces and torques have been established (34-35), the current orientation is preserved by aligning the side edge with the current value of 'SIDEEDGE', using the constant generation operator { }. The card is then moved slightly away from the connector, and constrained to a line parallel to the slot axis (38-40). A force is applied to move the edge of the card against the side of the small alignment ridge on the edge of the connector (41). Once touching the ridge (42), a servo is declared to null out the wrist torque about the slot normal (44-45).

At this time, the card is properly oriented and positioned along the slot axis. If everything is aligned, the card may simply be pushed into the slot. To check for partial insertion, a condition monitor is declared (49), which will asynchronously jump to the statement labeled 'inpart' (statement 70) if the corner moves .5 centimeter beyond the slot end in the insertion direction. The current corner location is saved as 'CENTER' (48), and the position is constrained to line which passes through CENTER, offset by the amount 'inc' in the direction of the slot normal. The constraining line is parallel to the insertion direction.

If the card does not go part way into the slot, a search procedure begins which moves the card away from the center point in .1 centimeter steps, alternating from side to side. The card is pushed toward the slot (52-53), the line position offset is changed (54-55), the card is pulled slightly away from the slot in case it is stuck (56-57), and the card is pushed down at its new location (58-59). This procedure is repeated with a negative offset (60-63). Each iteration, the offset is increased by .1 centimeter (65), up to a maximum of .5 centimeters (66). If the card still has not gone in part way, the program gives up (67-69). Remember that if at any time the card corner goes into the slot, the condition monitor will transfer control to 'inpart'.

AL also contains condition monitors, but they are more limited than those in RSS in the following way: AL pre-computes all trajectories and must have a planned value for the starting and ending points. Therefore, condition monitors which cause large deviations from planned values are forbidden. Since RSS does all its calculations in real time, using the current state of the arm, unplanned motions due to condition monitors cause no problems.

Once the card is in part way, it is necessary to wiggle it from side to side while pushing down. It is also necessary to align the hand directly over the slot since the search procedure may have left it offset somewhat. First, all position constraints are removed to allow compliance with external and applied forces (72). Next, the card is pushed toward the slot with a force of 70 newtons (73), and the component of force normal to the slot is specified to null out any torque about the slot axis. This procedure insures that the hand is above the slot and no bending force is applied to the card (75-76).

In addition to these two force servos, a third applies force along the slot axis to wiggle the card back and forth into the slot (78-81). Once the card is inserted to a depth of 1.2 centimeters (83), the card is wiggled two extra times, just to make sure that it's seated properly (84-85).

Once the card is in all the way, a force servo is declared to null out any external forces on the hand (86-87), the hand is opened to 2 centimeters (88-89), and the hand is moved 5 centimeters away from the slot, relative to its current position (90).

3.4 BOLT INSERTION WITH VISUAL TRACKING

This section describes a program which inserts a bolt into a hole, using dynamic visual tracking. To simplify the image processing, a white bolt is used, and the light is arranged so that the hole appears as a dark region on a gray surface. The hole is located once and assumed to not move, since it is obscured early in the insertion; however, the bolt is dynamically tracked and its position updated visually about 15 times per second. The program is shown in Figure 9, and the robot performing the task is shown in Figure 10A.

When the program begins, it is assumed that the bolt is already in the robot hand as shown in Figure 10B. Also, the programmer must explicitly define the insertion force 'f'. A value of 25 newtons is commonly used.

The functions 'BOLT' and 'HOLE' are declared to be vision functions (2), which means that RSS will expect their values to be determined by the vision processor. RSS is not concerned with the actual image processing algorithms, but expects the vision processor to contain a procedure for locating features given their name.

Only the positions of the bolt and the hole are determined visually. The hole axis is defined to be directed vertically upward (5), and the bolt axis is parallel to the robot fingers (6).

Next, RSS requests the vision processor to locate the hole (9) and waits until it is found (10). The hand is rotated so that the bolt axis is aligned with the hole axis, and the robot thumb is aligned so that the bolt will not be obscured by the robot fingers (12). The estimated

```
; Bolt insertion routine
; Clifford C. Geschke    9-12-78
          servo off                                                  (1)
          vision BOLT, HOLE                                          (2)
          print Insertion force = ;                                  (3)
          type f                                                     (4)
          define HOLEAXIS=[0,0,1]                                    (5)
          define BOLTAXIS=R$FINGER                                   (6)
          force ZERO                                                 (7)
          torque ZERO                                                (8)
          locate HOLE                                                (9)
          wait until flag found HOLE                                (10)
          print HOLE found.                                         (11)
; Move to above estimated hole location
          orient fixed BOLTAXIS;R$THUMB=-HOLEAXIS;[-1,0,0]          (12)
          define BOLTPOS=R$GRIP+2*BOLTAXIS                          (13)
          define BOLTGOAL=HOLE+2*HOLEAXIS                           (14)
          position point BOLTPOS=BOLTGOAL                           (15)
          servo on                                                  (16)
          wait until |BOLTPOS-BOLTGOAL|; lss .5;                    (17)
          define error=|(BOLTPOS-HOLE)#HOLEAXIS|                    (18)
; Locate and track BOLT
          trap 2 to lostit on flag lost BOLT                        (19)
          locate BOLT                                               (20)
          wait until flag found BOLT                                (21)
          track BOLT                                                (22)
          define BOLTPOS=BOLT                                       (23)
          position line BOLTPOS=HOLE;HOLEAXIS                       (24)
; Insert BOLT into HOLE
fwait:    force ZERO                                                (25)
          wait until error; lss .1;                                 (26)
          trap 1 to fwait on error; gtr .1;                         (27)
          force f*BOLTAXIS                                          (28)
          wait until R$FORCE.BOLTAXIS; lss -.8*f;                   (29)
          print All done!                                           (30)
          stop                                                      (31)
lostit:   servo off                                                 (32)
          forget BOLT                                               (33)
          print Lost BOLT                                           (34)
          stop                                                      (35)
```

Figure 9 - Bolt Insertion Program Using Visual Tracking

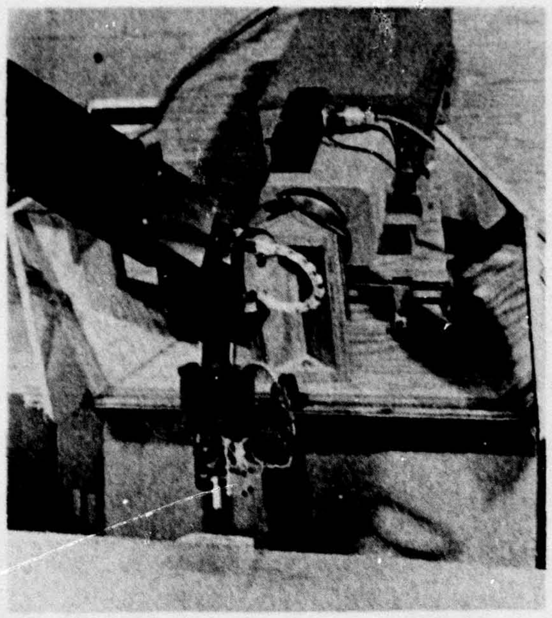
Figure 10A - Photograph of Bolt Insertion

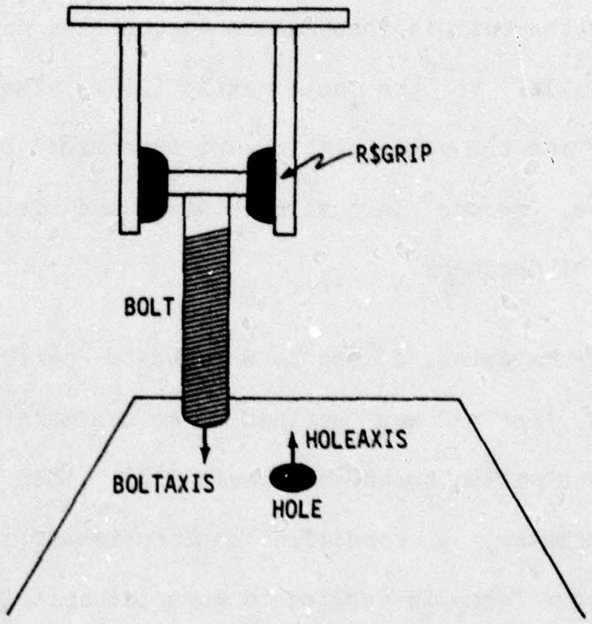

Figure 10B - Bolt Insertion Task

bolt position (13) is moved toward the point 2 centimeters from the visually located hole position (14-15), and the program waits until the positional error is less than .5 centimeters. Notice that the vision function 'HOLE' is used just like any other function.

Once the bolt is at this position, it is guaranteed to be within the TV camera's field of view. A condition monitor is declared to check for the possibility of the vision processor losing track of the bolt (19). A command is sent to the vision processor to locate the bolt, and once it is found, a command is sent to continually track it (20-22). The value of BOLT, and any function which depends upon it, are dynamically updated to reflect the most current bolt position. The function 'BOLTPOS' is now defined to be the visually determined function 'BOLT' (23), rather than the estimate made in statement 13.

Now the bolt is constrained to the line passing through the hole and parallel to the hole axis (24). The values of both the bolt location and the hole location are determined by the vision processor; therefore, errors in calibration of the vision system are unimportant because of feedback.

At this point, a loop is entered to perform the insertion. The function 'error' was defined to be the magnitude of the distance from the bolt position to the hole axis (18). When this error is less than .1 centimeter, a condition monitor is set to keep watching the error (27), and a force is applied to move the bolt into the hole (28). If the error ever becomes greater than .1 centimeter, the condition monitor transfers control to the statement labeled 'fwait', where the insertion force is cancelled (25) until the error is again acceptably small (26).

If the bolt is inserted completely into the hole, the opposing external force becomes large enough to satisfy the wait statement (29). In reality, the vision processor usually loses the bolt during insertion since it is partially obscured when it enters the hole.

3.5 CRANK TURNING USING FORCE CONTROL

The final example demonstrates how trivial crank turning actually is, if the problem is represented properly. There is no need to calculate elaborate trajectories or to require that the crank handle trace a circular path.

When the program begins, it is assumed that the robot hand is grasping the crank handle. Figure 11 shows the RSS program, Figure 12A shows the robot performing the task, and Figure 12B shows how the defined functions relate to the actual task.

Initially, the direction of the crank axis (2), the location of the crank center (3), and the magnitude of the cranking force (4) are defined. Next, any torque servo is cancelled (5), and all position constraints are removed, allowing the hand to move in compliance with any external or declared forces (6).

The hand orientation is fixed so that the fingers point vertically downward and the side of the hand is normal to the crank axis (7). The # operator indicates vector cross product.

Finally, a force servo is declared which applies a force of magnitude f, in the direction given by the cross product of the vector from the crank center to the robot gripper with the crank axis direction

```
; Crank turning program
; Clifford C. Geschke      11-30-78
        servo off                                          (1)
        define AXIS=[0,1,0]                                (2)
        define CENTER=[-10,50,20]                          (3)
        define f=40                                        (4)
        torque ZERO                                        (5)
        position none                                      (6)
        orient fixed R$FINGER;R$THUMB#R$FINGER=[0,0,-1];AXIS (7)
        force f*< (R$GRIP-CENTER)#AXIS >                   (8)
        servo on                                           (9)
```

Figure 11 - Crank Turning Program Using Force Control

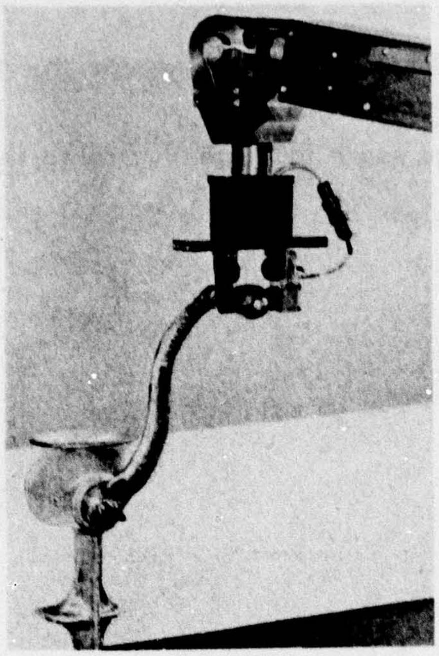

Figure 12A - Photograph of Crank Turning

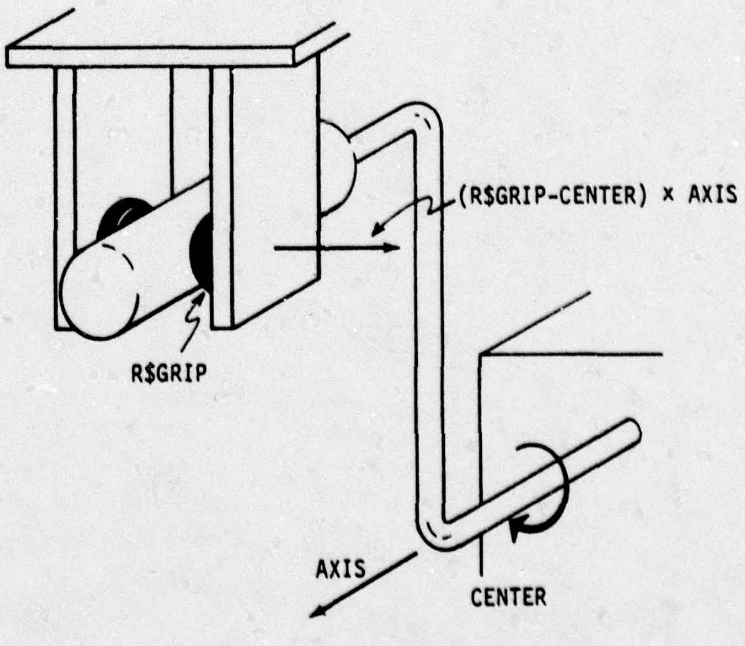

Figure 12B - Crank Turning Task

vector (8). The < > operator returns the direction of a vector.

This force servo will apply a force tangent to the crank trajectory at any point along the trajectory, once the servo is activated (9).

4. CONCEPTS AND REPRESENTATION

This section briefly presents RSS from a programmer's point of view. It describes how sampled-data servos and condition monitors are declared, and provides some simple examples of how they are used. RSS is intended to allow powerful sensor-driven manipulator programs to be written in a manner which appeals to the programmer's intuition. For a more complete explanation of RSS program syntax, see Appendix D.

4.1 VECTORS AND SCALARS

All positions in RSS are expressed as 3-element vectors which represent points in a 3-dimensional Cartesian coordinate system. The units for position vectors are centimeters. Directions are expressed as 3-element vectors with unit magnitude which refer to the same Cartesian coordinate system. RSS deals only with these 3-element vectors and with scalars. The robot coordinate system has its origin at the center of the robot base at table height. X is measured along the length of the table, Y is measured along the breadth of the table, and Z is measured vertically.

4.2 MANIPULATOR STATE REPRESENTATION

The manipulator state is accessed via system-defined functions. As shown in Figure 3, R$GRIP is the name of a function which always evaluates to the position of the robot gripping point, the function R$FINGER evaluates to the direction of the robot fingers, R$THUMB evaluates to the direction from finger one to finger two, and r$hand evaluates to a scalar containing the hand opening.

4.3 POSITION DECLARATIONS

Position servos are declared by specifying a reference point and a goal for that point. For example, the declaration:

```
position point R$GRIP=[-10,40,50]
```

will establish a servo to bring the robot gripper to the point (-10,40,50) in the robot coordinate system. If the robot is active, it will immediately begin moving to the point. Once there, it will oppose any force which would move it from that point. This declaration may also be viewed as completely constraining the position of the gripper. Other declarations are:

```
position line R$GRIP=[-10,40,50];[0,0,1]
```

which constrains the gripper to the line which passes through (-10,40,50) and is parallel to the direction (0,0,1), and:

```
position plane R$GRIP=[-10,40,50];[0,1,0]
```

which constrains the gripper to the plane which passes through the point (-10,40,50) and is normal to the direction (0,1,0). The declaration:

```
position none
```

specifies that the position is to be completely unconstrained. The hand is free to move in any unconstrained degrees of freedom in compliance with declared or external forces.

4.4 EXPRESSIONS AND USER FUNCTIONS

Declarations may be much more complex, using expressions and user-defined functions. Expressions are formed of functions and constants combined with various operators. The operators are: addition (+), subtraction (-), scalar multiplication (*), scalar division (/), vector dot product (.), vector cross product (#), vector magnitude or scalar absolute value (| |), and vector direction (< >). User functions are defined by expressions. For example,

```
define DRILL=R$GRIP+10*R$FINGER
```

defines a function named 'DRILL' which evaluates to a point 10 centimeters from the robot grip position, in the direction of the robot fingers. The value of DRILL is continually updated to reflect any changes in R$GRIP or R$FINGER. To save the current value of a function, the immediate evaluation operator ({ }) is used. The statement:

```
define HOLE={DRILL}
```

defines a function named 'HOLE' which evaluates to a constant equal to the value of DRILL when the statement was executed. In general, servo declarations may use arbitrary expressions involving any user-defined or system-defined functions. The declaration:

```
position point DRILL=HOLE+[0,0,5]
```

will constrain the point determined by DRILL to a point 5 centimeters above the point determined by HOLE.

Even when an active servo is referencing a user-defined function, it is possible to change its definition. The servo will then behave according to the new definition.

4.5 EXTERNAL SENSORS AND VISION

Data from external sensors is expressed as dynamically changing functions, just as functions which refer to the robot state. For example, if the vision processor is tracking an object named 'BOLT', any reference to BOLT will return the most current value of its location as determined by the TV camera. The statement:

```
vision BOLT
```

declares to RSS that the value of the function BOLT will be determined by the vision processor. The statement:

```
locate BOLT
```

requests the vision processor to visually determine the location of BOLT and update the function accordingly. The vision processor must contain information which associates the name 'BOLT' with a routine to locate it. RSS itself is not concerned with how objects or features are located. If the location of BOLT does not change, the declaration:

```
position point R$GRIP=BOLT
```

is sufficient to move the robot gripper to the bolt location. If the location may change, it is necessary to add the statement:

```
track BOLT
```

which will cause the vision processor to continually update the value of BOLT based upon the stream of images it is processing. BOLT is then dynamically updated like a system-defined function.

4.6 PURE VISUAL FEEDBACK

The previous position declaration depends upon accurate calibration between the vision system and the robot system. The location of BOLT as determined visually must be very close to its actual location. Calibration requirements may be relaxed if the vision processor is also able to track the robot hand, which means that both the reference position and the goal position can be determined visually. Assume there is a hand mark which the vision processor can see, then the statements:

```
vision HANDMARK
locate HANDMARK
track HANDMARK
position point HANDMARK=BOLT
```

will establish a servo which calculates its position error based entirely upon visual data. Since the vision system is included entirely within the feedback loop, small errors in calibration will be eliminated.

4.7 ORIENTATION DECLARATIONS

Orientation is specified as the alignment of direction vectors. For example, suppose it is desired to orient the hand so that the fingers are directed vertically downward. The statement:

```
orient align R$FINGER=[0,0,-1]
```

declares a servo which will rotate the hand to the desired orientation, and oppose any torques which attempt to move it from that orientation. Notice that the rotation about the finger axis is unconstrained. The orientation may be completely constrained, or fixed, by specifying the alignment of two pairs of vectors. The statement:

```
orient fixed R$FINGER;R$THUMB=[0,0,-1];[1,0,0]
```

declares a servo which not only aligns the finger direction with the vector (0,0,-1) as before, but also aligns the R$THUMB vector with the X axis (1,0,0). Hand orientation may be completely unconstrained by the declaration:

```
orient none
```

The hand is free to rotate in any unconstrained degrees of freedom in compliance with declared or external torques. As in position declarations, arbitrary vector expressions may be used as any of the arguments in orientation declarations.

4.8 FORCE AND TORQUE DECLARATIONS

Force and torque servos may also be specified in RSS. These servos are considered secondary to position and orientation servos, and will only apply components which cause motion in unconstrained degrees of freedom. For example, if the position is constrained to a line, only components of force parallel to that line are recognized. Orientation and torque servos interact in a similar way. The RSS statement:

```
force VECTOR
```

will establish a servo to exert a force at the robot wrist in the direction of VECTOR and with a magnitude equal to the magnitude of VECTOR. The units of force are newtons. The statement:

```
torque VECTOR
```

will establish a servo to exert a torque at the robot wrist about an axis in the direction of VECTOR and with a magnitude equal to the magnitude of VECTOR. The units of torque are newton-meters. As in other declarations, any vector expression can be used as an argument.

The force and torque servos are open loop. The vectors specified are converted into robot joint torques as determined by the manipulator position, and written to the joints. If there were external force sensors on the robot, it would be simple to specify a feedback servo.

4.9 VELOCITY CONTROL

Velocity control is useful in many applications such as painting or welding. It has been implemented in RSS as servoing to a position which varies as a function of time. To permit the definition of time-dependent functions, a clock declaration is included which indicates that the value of a scalar function is to be dynamically incremented to reflect the passage of time. The statement:

```
clock foo=20
```

declares a scalar function named 'foo' which is initially set to 20, and which will evaluate to 20 plus the number of seconds which have passed since the declaration was made. The statements:

```
clock foo=0
position point R$GRIP=HOLE-2*foo*[0,1,0]
```

declare a position servo which moves the robot gripper away from the point given by HOLE at the rate of 2 centimeters per second, in the -Y direction.

When velocity is implemented in this manner, force servos are limited to degrees of freedom not constrained by the position declaration, and the manipulator exerts whatever force is necessary to maintain the velocity. In essence, velocity control is implemented as a spatial aspect of manipulation.

4.10 CONDITION MONITORS

Once a servo is declared, the robot will begin to move accordingly. It is necessary to know when a certain action is complete so that the next program step may begin. The definition of completion depends upon the task being performed. Some gross motions are considered complete when the reference is within 2 centimeters of the end point, while fine motions may require that the error be less than one millimeter. In a touching operation, completion consists of establishing a certain force. When pouring a liquid, completion may occur when the vision system detects the appropriate liquid level.

In RSS, completion conditions are specified by wait statements and condition monitor declarations. A wait statement causes the program execution to be suspended until an arithmetic relation is satisfied or certain flags are set. Although the program is suspended, the declared servos are still active and the robot continues moving. By using the appropriate arithmetic relation, varied completion conditions are easily specified. For example, the statement:

```
    wait until |DRILL-HOLE|; lss 0.2;
```

will suspend the program until the magnitude of the vector from HOLE to DRILL is less than 2 millimeters.

Condition monitors continually check for certain arithmetic relations to be satisfied or certain flags to be set. If the condition is satisfied, control is asynchronously transfered to a specified program statement. Condition monitors are declared with the trap statement. For example:

```
    trap 1 to match on R$TORQUE.AXIS; gtr 4;
```

will establish a condition monitor with priority 1 which will jump to the statement labeled 'match' if the external torque about AXIS is greater than 4 newton-meters.

4.11 PROGRAM CONTROL

Program control is straightforward. The normal sequential execution of program statements may be altered by an unconditional or conditional branch, or by condition monitor firing. Conditional branches may evaluate an arithmetic relation or test certain flags, just as wait and trap statements. Other statements allow output to the user terminal and suspension of the program until the user types 'continue'.

By unifying the representation of data from both built-in and external sensors, by allowing manipulator actions to be specified in terms of sampled-data servos, and by expressing the manipulator state in terms of easily understood vectors and scalars, RSS facilitates the programming of sensor-based robots and provides an intuitive method of describing manipulator tasks.

5. SYSTEM ORGANIZATION

This section briefly describes the internal workings and organization of RSS. A detailed description of the actual RSS software is given in Appendix F. Figure 13 is a block diagram of the system.

5.1 HARDWARE DESCRIPTION

The manipulator controlled by RSS is a Stanford electric arm, shown in Figure 1. It has six joints which allow the hand to achieve any position and orientation within a one meter radius of the base. Each joint contains an electric motor, a brake, a position sensor, and a velocity sensor. The hand has two fingers which move together toward a center point. On one finger is a touch sensor which measures the squeezing force of the hand [26]. All of the joints and the hand are controlled by a Hardware Servo which supplies sufficient velocity feedback to maintain stability, even with a sampling rate of 15 Hz. Associated with each joint are two 12-bit registers for reading its position and velocity, and a 12-bit register for writing its motor current. All these registers are calibrated so that the actual position, velocity, and torque are known for each joint, along with the effect of the velocity feedback.

5.2 SOFTWARE DESCRIPTION

The User Command Processor accepts commands from the user terminal or from a specified disk file. These commands may be editing commands to create or modify RSS programs, program control commands to start or stop RSS program execution, or servo commands.

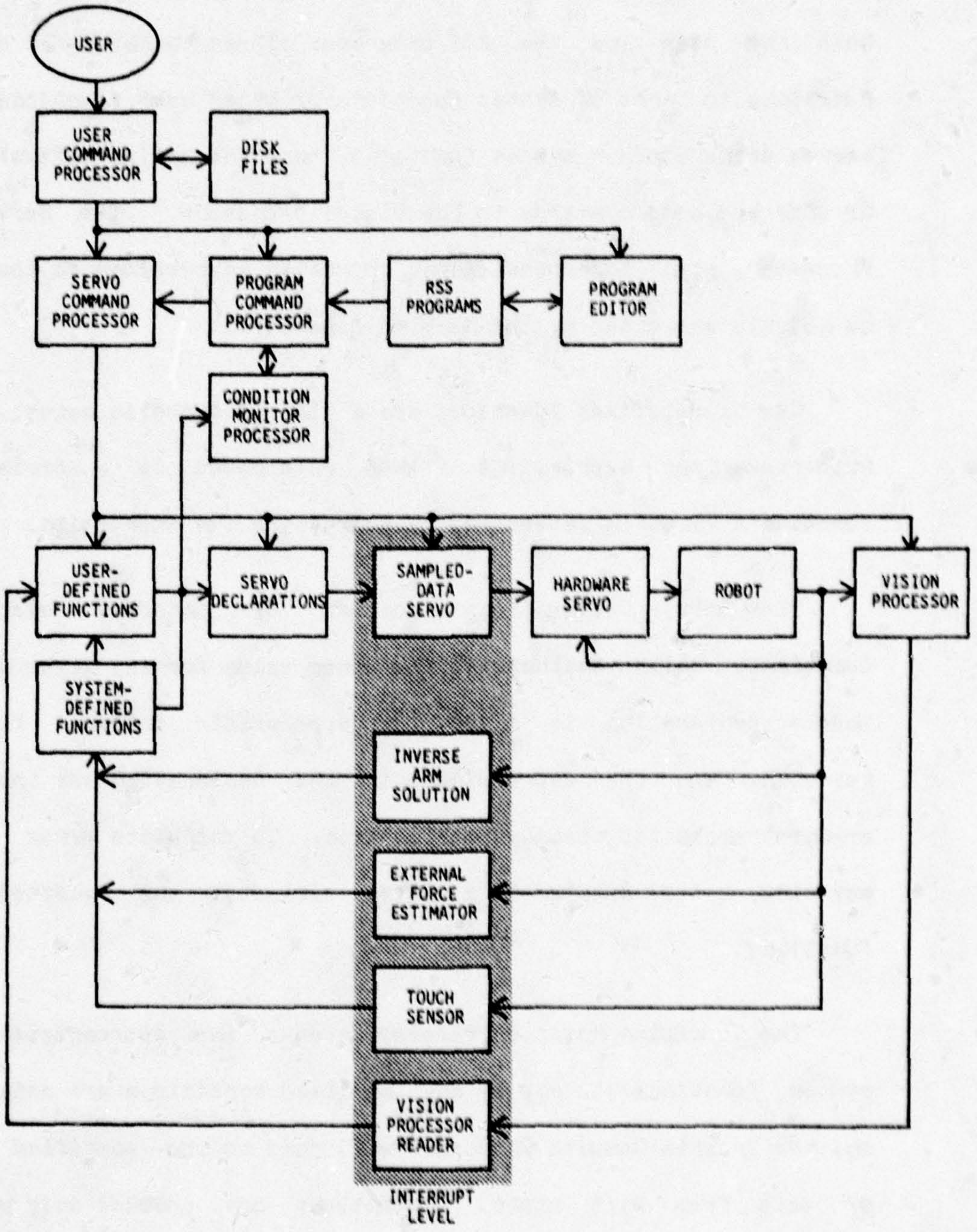

Figure 13 - RSS Block Diagram

Servo commands are received by the Servo Command Processor from both the user and the RSS programs. These commands may define user functions in terms of system functions or other user functions, declare servos using user or system functions, turn the entire software servo on or off, and send commands to the vision processor. The Servo Command Processor also tokenizes vector or scalar expressions so that they may be quickly evaluated by the Sampled-Data Servo.

The User-Defined Functions are a list of symbolic names associated with tokenized expressions. When referenced in a declaration, a function's value is determined by evaluating its expression.

The Servo Declarations consist of internal system-defined functions, which evaluate to the error value for any given servo mode. When a declaration is made, the appropriate internal function is selected, and the expressions in the declaration are inserted into argument slots for that system function. To calculate error terms for servoing, the Sampled-Data Servo evaluates the selected internal functions.

The Condition Monitor Processor checks the appropriate user and system functions to see if any specified conditions are satisfied. If so, the Program Command Processor will jump to the specified statement or exit from wait state. Conditions are checked only when an RSS program is in wait state, or when a branch statement is executed.

Vision processor commands are simply sent to the PDP10 via a high speed data link.

5.3 INTERRUPT LEVEL SOFTWARE

The Sampled Data Servo runs at interrupt level, and uses the servo declarations to calculate the robot joint torque values which are written to the hardware servo. A sampling rate of only 15 Hz. is used because of the large amount of computation performed for each sample, and because data from the vision processor cannot be acquired any faster.

The values of the various System-Defined Functions are determined at interrupt level. At each sample time: the Inverse Arm Solution determines, in Cartesian coordinates, the position of the robot wrist and the orientation of the robot hand; the External Force Estimator uses its knowledge of the joint motor current, velocity, and hardware velocity feedback to estimate the torque on each joint and transform these torques to a wrist force and hand torque vector in Cartesian space; the Touch Sensor reads the finger touch sensor value; and the Vision Processor Reader reads any messages from the Vision Processor, updating the values of user-defined vision functions.

Appendix A describes the arm solution algorithm, and Appendices B and D describe the methods used to estimate the external force and torque.

The actual sequence of operations during each interrupt is as follows:

1. The current wrist position, hand orientation, wrist force, and wrist torque are read from the robot joint sensors and converted into Cartesian 3-vectors. Also, data from external sensors and the vision processor is read.

2. The position and orientation errors are computed according to the servo declarations. The desired position and orientation are calculated and converted to joint angles.

3. The desired force and torque vectors are computed from the servo declarations and converted to joint torque values.

4. Gravity and friction compensation are computed and converted to joint torque values.

5. A velocity for each joint is computed so as to move from the current position to the desired position by the next sample time. These velocities are converted to pseudo-torques by using the knowledge about the hardware velocity feedback for each joint.

6. All the joint torque components (from 3, 4, and 5 above) are added and the velocity components are then scaled so that no joint exceeds its maximum allowed torque. This procedure has the effect of forcing all joints to track the slowest joint.

7. The joint torques are written to their respective motor current registers.

An interlock is provided to avoid instability in case external sensors (especially the vision processor) fail to provide data as fast as expected. If necessary external data is not available during a servo calculation, the results of the last valid servo calculation are retained. Also, the velocity of the joints is reduced in proportion to the number of sampling intervals since the last valid external data was read. Thus, the manipulator performance degrades gracefully if an external sensor slows down.

5.4 THE VISION PROCESSOR

The organization of the vision processor hardware and software is shown in the block diagram in Figure 14. The vision processor consists of a TV camera which is interfaced to a PDP10 computer. The blocks in the diagram show how the hardware and the various portions of the PDP10 software are interconnected.

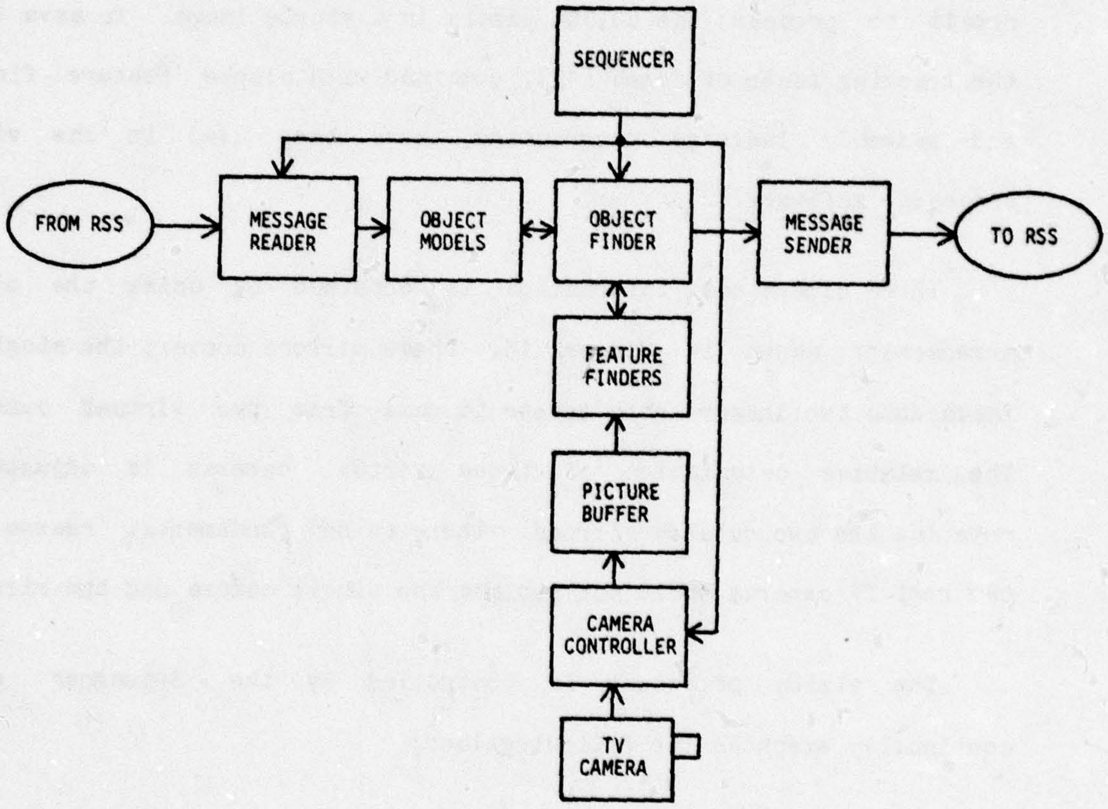

Figure 14 - Vision Processor Block Diagram

The vision hardware consists of a single vidicon TV camera interfaced to a PDP10 computer [27]. A single image of 252 by 238 pixels with 64 gray levels is obtained. The average time to read an image into PDP10 memory is about 30 milliseconds. Since RSS expects servoing data at the rate of 15 Hz., little more than 30 milliseconds remain to process the 60,000 pixels in a single image. To save time, the tracking ideas of Jones [19], combined with simple feature finders and assembly language programming, have been used in the vision processor software.

Three dimensional information is obtained by using the mirror arrangement shown in Figure 15. These mirrors convert the single TV image into two images which appear to come from two virtual cameras. The relative orientation of these virtual cameras is adjusted by rotating the two outside mirrors. There is no fundamental reason why two real TV cameras could not replace the single camera and the mirrors.

The vision processor is controlled by the Sequencer which continually executes the following loop:

1. The vision processor job sleeps until a message is received from RSS.
2. The Message Reader reads all messages from RSS and takes the appropriate action.
3. If no vision requests are pending, go to step 1.
4. If vision requests are pending, the Camera Controller reads a picture from the TV camera.
5. The Object Finder attempts to locate the position of each active object and sends the appropriate messages to RSS.
6. When the Object Finder is done, an end-of-cycle message is sent to RSS.

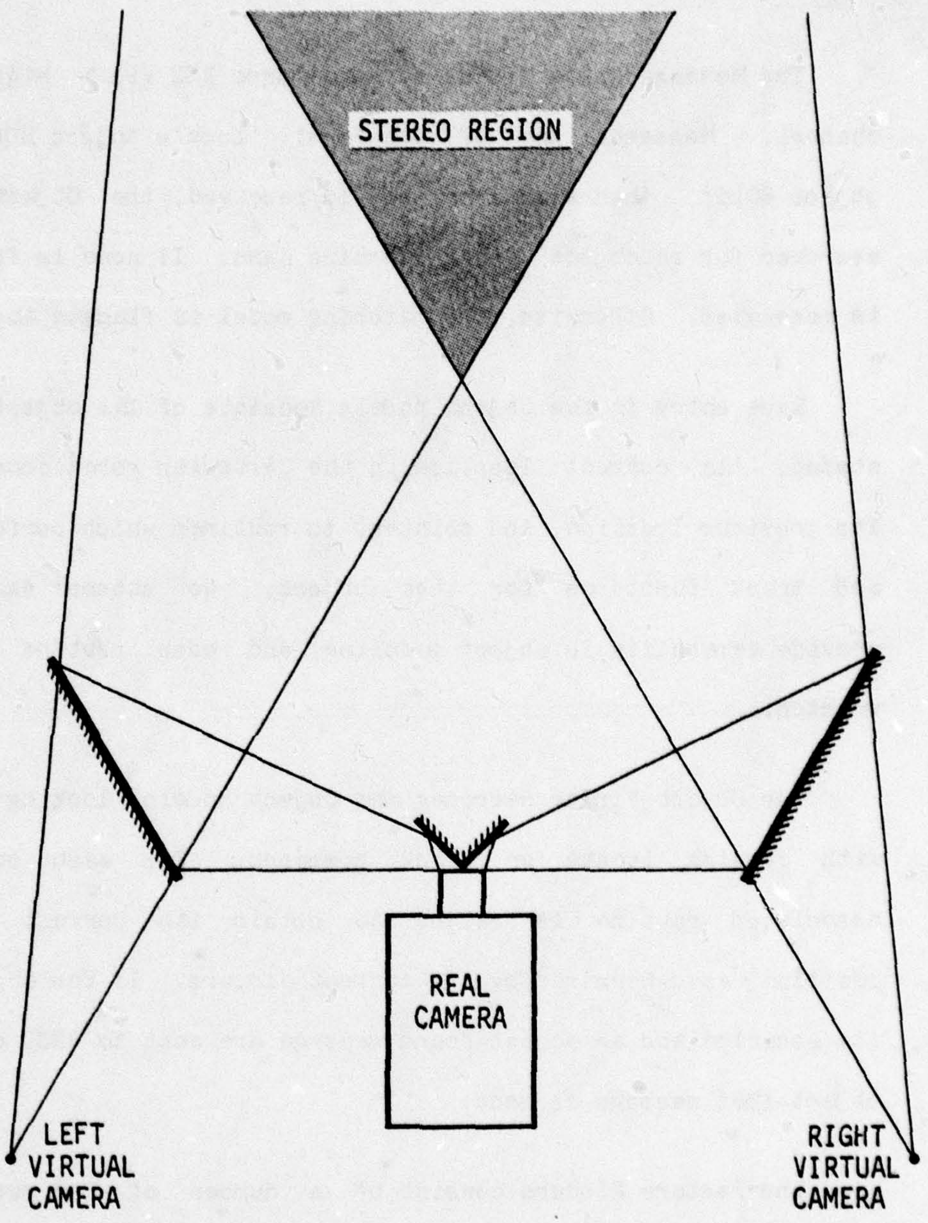

Figure 15 - Mirror Arrangement for Stereo Vision

7. Go to step 2.

The Message Reader reads messages from RSS via a high speed data channel. Messages are of the form: 'Locate object HOLE,' or 'Track object BOLT'. When such a message is received, the Object Models are searched for an object with a matching name. If none is found, an error is generated. Otherwise, the matching model is flagged appropriately.

Each entry in the Object Models consists of the object's name, its status, its current location in the Cartesian robot coordinate space, its previous location, and pointers to routines which perform the locate and track functions for that object. No attempt has been made to provide generality in object modeling, and each routine is specially written.

The Object Finder searches the Object Models looking for objects with pending locate or track commands. For each one found, the associated routine is called to obtain its current 3-dimensional position as determined by the current picture. If the object is found, its location and an object-found message are sent to RSS, otherwise, an object-lost message is sent.

The Feature Finders consist of a number of 2-dimensional image processing routines which are called by the special purpose object finding routines defined by the Object Models. Features are located in both the right and left halves of the stereo image, and their positions in 3 dimensional Cartesian robot coordinates are calculated by performing the coordinate transformations as described in Appendix C. The Feature Finders deal with binary images obtained by setting a variable threshold for the gray-scale image. The routines allow a program to easily perform the following picture operations:

1. Scan the entire image for a point below or above a threshold, or a minimum or maximum point.
2. Spiral search from a given center for a given radius, looking for a point above or below a threshold.
3. Trace the outline of a region and calculate its area, perimeter and center.
4. Grow a region and calculate its area, perimeter, center and average value.

Using the operations described above, routines have been written to locate a light or dark region, given its estimated position, minimum and maximum area bounds, and area to perimeter-squared ratio.

The threshold needed to define the proper region boundary is dynamically adjusted so that tracking may continue when the light intensity changes due to motion in the scene.

6. CONCLUSIONS

Experience with RSS has shown that the concepts which it embodies are useful in programming a sensor-based robot manipulator. Programs involving external sensors may be written as easily as those using built-in sensors. The unified method of handling sensory data, and the intuitive representation of position and direction in a Cartesian coordinate system, have not been attained by sacrificing programming flexibility. In fact, since RSS allows the spatial and kinesthetic aspects of manipulation to be freely combined, permits arbitrary sensors to be used for controlling any such aspects, and enables the completion conditions for each program step to be specified, it provides more flexibility than any other manipulator control scheme.

The following sections describe how the various components of RSS could be improved, and suggests the direction future research should take.

6.1 SAMPLED-DATA SERVO

The most severe limitation of the current RSS implementation results from the 15 Hz. sampling rate. This extremely slow rate requires that the robot move very slowly to maintain stability. The maximum manipulator hand velocity is about 10 centimeters per second. The main reason for using this sampling rate is that the servo calculations are performed by an interpreter using floating point arithmetic on a PDP11/40 minicomputer. When several complicated servos are active, as much as 50 milliseconds is used out of the 67 millisecond interval between samples. Minicomputers are available which perform floating point operations almost two orders of magnitude faster than the PDP11/40. Such a computer would greatly improve the performance of this system.

High speed robot operation will also require the servo scheme to handle the dynamic aspects of manipulation. The current robot model totally ignores the effects of acceleration, inertia, and Coriolis force. These dynamic effects couple between the joints and make servo calculation extremely complicated. Raibert and Horn have proposed some solutions to the computational difficulties of handling the dynamic aspects of control in real time [28].

6.2 FORCE AND TORQUE MEASUREMENT

Forces and torques are estimated based upon a static model of the robot joints. Forces due to acceleration and coupling between joints are therefore included in these estimates. If a dynamic arm model were used, the estimates could be somewhat refined. Even at slow speed, the estimates are inaccurate because of the large amount of friction in the various manipulator joints. The Stanford arm uses harmonic drives in its main joints which seem to contribute a significant friction which varies with position. This friction is also included in the estimated external force and torque, and is severe enough to cause a force as large as 15 newtons (3.4 pounds) during a smooth motion through free space. The only solutions to this problem are to redesign the robot manipulator to minimize joint friction, or to add special force sensors to the robot joints or hand.

6.3 VELOCITY CONTROL

The method of velocity control in this implementation of RSS consists of servoing to a position which is a function of time. In this case, force servos are limited to degrees of freedom not specified by the position servo, and the manipulator exerts whatever force is necessary to maintain velocity. An alternative method of velocity control would be to provide a function which evaluates to the manipulator hand or wrist velocity, and to declare a force servo in terms of the difference between the actual velocity and some desired velocity. In this case, velocity servos can cause motion only along degrees of freedom not specified by a position servo, and the maximum force exerted by the manipulator is determined by the force declaration. This case can also be viewed as a method of limiting the manipulator velocity while attempting to establish a certain force. Since both methods of velocity control have some value, it would be reasonable for a future implementation to include them both.

6.4 PROGRAM CONTROL

As mentioned before, RSS is not designed to provide the capabilities of a general purpose computer programming language. Its purpose is to demonstrate some fundamental concepts in manipulator programming and to provide a test bed for those concepts. Nevertheless, it does become tedious to specify the details of each trajectory segment and the completion conditions for each program step. A practical RSS should allow macro definitions and subroutines. In fact, it should probably be combined with a high level block-structured language such as ALGOL or PL/1. Frame declarations and coordinate transformations such as in AL or WAVE could also be useful.

6.5 COMPUTER VISION

Computer vision still requires too much time to be useful for dynamic visual servoing. The maximum rate at which the RSS vision processor can examine pictures is about 15 per second, using very simple algorithms, hand-coded in assembly language and running on a PDP10. Such a scheme is entirely unsatisfactory for controlling a high speed manipulator in a real application. What is needed is the development of image processing hardware which can quickly generate a concise representation of non-trivial features in a scene. In the absence of such a breakthrough, the following observations can still be made:

1. Using current techniques, it only takes a few seconds to visually locate a stationary object with sufficient accuracy to move the manipulator close to it.

2. When the manipulator is close to an object, it will probably be moving slowly in order to establish some kinesthetic relation. Therefore, it may be possible to use a slow visual sampling rate to control the final spatial relations.

6.6 MULTIPLE MANIPULATOR COORDINATION

Multiple manipulator coordination can improve a robot system and extend its capabilities. Since one arm can act as a jig for another arm, the number of special purpose jigs can be reduced, and the time required to insert and remove parts from a jig can be eliminated. Also, many tasks involve parts which are too large or awkward for a single manipulator. Several arms may be required when handling non-rigid parts or performing simultaneous operations.

RSS provides an excellent environment for multi-arm coordination. To a particular manipulator, information about other manipulators can be represented as data from external sensors. In fact, those other manipulators are external sensors. Since RSS permits declaring servos in terms of arbitrary sensory information, the following modes of coordination are easily specified:

1. Each arm performs an independent task.
2. All arms servo using a common sensor signal.
3. The arms track the position of a master arm.
4. The arms track each other through the forces transmitted by a common workpiece.

One method of implementing coordination would be to have each arm controlled by an independent computer while storing all manipulator state and external sensor information in a shared memory. Of course, protocol details would have to be worked out.

It should be noted that the present RSS program control scheme would permit multiple manipulator programs since all declared servos are assumed to run in parallel. Such a program would declare servos for all the arms and then specify condition monitors and wait until certain conditions were satisfied. The conditions could be in terms of any or all of the different manipulators. If the shared memory approach were used, a separate processor could be used to evaluate condition monitors.

6.7 FUTURE IMPLEMENTATIONS

This final section speculates about how a future RSS might be implemented. The system envisioned is described below.

In the interest of achieving total independence from a particular manipulator configuration, all joint level servoing is replaced by servoing in a 3-dimensional Cartesian coordinate system. The entire state of the manipulator (including position, velocity, acceleration, external forces, and inertia) is transformed to this coordinate system. State feedback gains due to the physical manipulator and control electronics are also transformed. Using all this information, a desired force and torque are calculated and transformed from Cartesian coordinates to actual joint torques.

It may be argued that such transformations are costly and time consuming, but the current trends indicate that computers will continue to become more powerful and less expensive. It is possible that robot manufacturers would supply a hardware device with the robot which performs the necessary transformations.

In the proposed scheme, RSS servo declarations consist of nothing more than selecting the appropriate control inputs and feedback gains. The declarations for various aspects of manipulation consist of macros or subroutines which supply the appropriate servo parameters. Not only does such a scheme offer manipulator independence, but it also suggests a way to apply modern control methods to an otherwise intractable problem.

Eventually, the control of robot manipulators will be limited by our inability to accurately represent the overall manipulation process. By identifying some of the problems in robot manipulation, and by proposing some solutions, it is hoped that this thesis has provided some insights into how future robot systems should be designed.

REFERENCES

1. Bejczy, A. K., "Issues in Advanced Automation for Manipulator Control," _Proceedings of the Joint Automatic Controls Conference_, Purdue University, pp. 700-711, July 1976.

2. Scheinman, V., _Design of a Computer Controlled Manipulator_, Stanford Artificial Intelligence Memo AIM-92, Stanford University, June 1969.

3. Engelberger, J., _Stand Alone Versus Distributed Robotics_, Symposium on Computer Vision and Sensor-Based Robots, General Motors Research Laboratories, Warren, Michigan, September 1978.

4. Lynch, P. M., _Economic-Technological Modeling and Design Criteria for Programmable Assembly Machines_, Mechanical Engineering Department Ph.D Thesis, Massachusetts Institute of Technology, June 1976.

5. Nevins, J., Whitney, D., Drake, S., et al., _Exploratory Research in Industrial Modular Assembly_, Fourth Report prepared for the National Science Foundation covering the period from September 1, 1975 to August 31, 1976.

6. Abraham, R., "Programmable Automation of Batch Assembly Operations," _The Industrial Robot_, vol. 4, no. 3, pp. 119-131, September 1977.

7. Wang, S. and Will, P., "Sensors for Computer Controlled Mechanical Assembly," _The Industrial Robot_, vol. 5, no. 1, pp. 9-18, March 1978.

8. Bolles, R. and Paul, R., _The Use of Sensory Feedback in Programmable Assembly Systems_, Stanford Computer Science Report STAN-CS-396, Stanford University, October 1973.

9. Watson, P. and Drake, S., "Pedestal and Wrist Force Sensors for Automatic Assembly," _Proceedings of the Fifth International Symposium on Industrial Robots_, ITT Research Institute, Chicago, Illinois. pp. 501-511, September 1975.

10. Whitney, D., "Force Feedback Control of Manipulator Fine Motions," _Proceedings of the Joint Automatic Controls Conference_, Purdue University, pp. 678-693, July 1976.

11. Nevins, J. and Whitney, D., "Industrial Assembly Parts Mating Studies," _Proceedings of the Sixth NSF Grantees' Conference on Production Research and Technology_, Purdue University, September 1978.

12. Paul, R., _Modelling, Trajectory Calculation, and Servoing of a Computer Controlled Arm_, Stanford Computer Science Report STAN-CS-72-311, Stanford University, November 1972.

13. Paul, R., WAVE: A Model-Based Language for Manipulator Control, Society of Manufacturing Engineers Technical Paper MR76-615, 1976.

14. Paul, R. and Shimano, B., "Compliance and Control," Proceedings of the Joint Automatic Controls Conference, Purdue University, pp. 694-699, July 1976.

15. Finkel, R., Taylor, R., Bolles, R., Paul, R., and Feldman, J., AL, A Programming System for Automation, Stanford Computer Science Report STAN-CS-74-456, Stanford University, November 1974.

16. Gill, A., Visual Feedback and Related Problems in Computer Controlled Hand Eye Coordination, Stanford Computer Science Report STAN-CS-72-312, Stanford University, October 1972.

17. Rosen, C., et al., Machine Intelligence Research Applied to Industrial Automation, First through Sixth Reports prepared for the National Science Foundation covering the period April 1, 1973 to October 31, 1976.

18. Birk, J., Kelley, R., et al., General Methods to Enable Robots with Vision to Acquire, Orient, and Transport Workpieces, Fourth Report prepared for the National Science Foundation covering the period August 15, 1977 to July 15, 1978.

19. Jones, V. C., Tracking: An Approach to Dynamic Vision and Hand-Eye Coordination, Coordinated Science Laboratory Report R-696, University of Illinois at Urbana-Champaign, December 1975.

20. Rosen, C., et al., Machine Intelligence Research Applied to Industrial Automation, Eighth Report prepared for the National Science Foundation covering the period August 1, 1977 to July 31, 1978.

21. Bolles, R., Verification Vision within a Programmable Assembly System, Stanford Computer Science Report STAN-CS-77-591, Stanford University, December 1976.

22. Finkel, R., Constructing and Debugging Manipulator Programs, Stanford Computer Science Report STAN-CS-76-567, Stanford University, August 1976.

23. Taylor, R., The Synthesis of Manipulator Control Programs from Task-Level Specifications, Stanford Computer Science Report STAN-CS-76-560, Stanford University, July 1976.

24. Lozano-Pérez, T., The Design of a Mechanical Assembly System, Artificial Intelligence Laboratory Report AI-TR-397, Massachusetts Institute of Technology, December 1976.

25. Lieberman, M. and Wesley, M., AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1976.

26. Geschke, C. C., "A Variable Capacitance Touch Sensor," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Massachusetts Institute of Technology, p. 772, August 1977.

27. Snyder, W., Jones, V., A User's Guide to the CSL Vision System, Coordinated Science Laboratory Report R-714, University of Illinois at Urbana-Champaign, January 1976.

28. Raibert, M. and Horn, B., "Manipulator Control Using the Configuration Space Method," The Industrial Robot, vol. 5, no. 2, pp. 69-73, June 1978.

29. Duda, R. and Hart, P., Pattern Classification and Scene Analysis, New York: John Wiley & Sons, pp. 398-401, 1973.

APPENDIX A

This appendix contains a description of the arm solution used by RSS. This method of solution was invented by David Monck of the Coordinated Science Laboratory, developed and implemented by the author.

In the following equations, upper case variables denote three element column vectors, and lower case variables denote scalar quantities. The robot's rotational joints are denoted j1 through j5, where j1 refers to the main joint which rotates the entire robot arm about a vertical axis, and j5 refers to the last joint on the robot wrist. The boom extension is treated separately as b. The robot finger spacing is not considered part of the arm solution. Figure 16 shows how the symbols mentioned relate to the actual manipulator.

All positions are in robot coordinates. The robot coordinate system is a Cartesian system with the origin at the center of the robot base, at table height. The X axis is directed horizontally along the length of the robot table, the Y axis is directed horizontally along the breadth of the robot table, and the Z axis is directed vertically upward.

Notation:

t[n]	denotes the angular position of rotational joint j[n]
A[n]	denotes a vector of unit magnitude which points along the axis of rotational joint j[n].
b	denotes the distance from the j2 axis to the robot wrist.
W	denotes the position of the robot wrist. Its components are denoted wx, wy, and wz. The wrist is defined to be at the intersection of the j3, j4, and j5 axes.
H	denotes the position of the robot hand. The hand is defined to be the point half way between the two robot finger pads.

Some constants:

k1	is defined to be the distance from the robot table to the j2 axis.
k2	is defined to be the distance from the j1 axis to the center of the boom.
k3	is defined to be the distance from the robot wrist to the robot hand, along the j5 axis.

A.1. THE INVERSE ARM SOLUTION

To solve for the robot wrist and hand positions, given the joint angles, the A vectors must be determined. Define A0 to be a unit vector in the Y direction. A1 is a unit vector in the Z direction.

The rest of the A vectors may be calculated using the formula:

```
A[n] = A[n-2]*cos(t[n-1])
          + A[n-1] x A[n-2]*sin(t[n-1])                    (A.1)
```

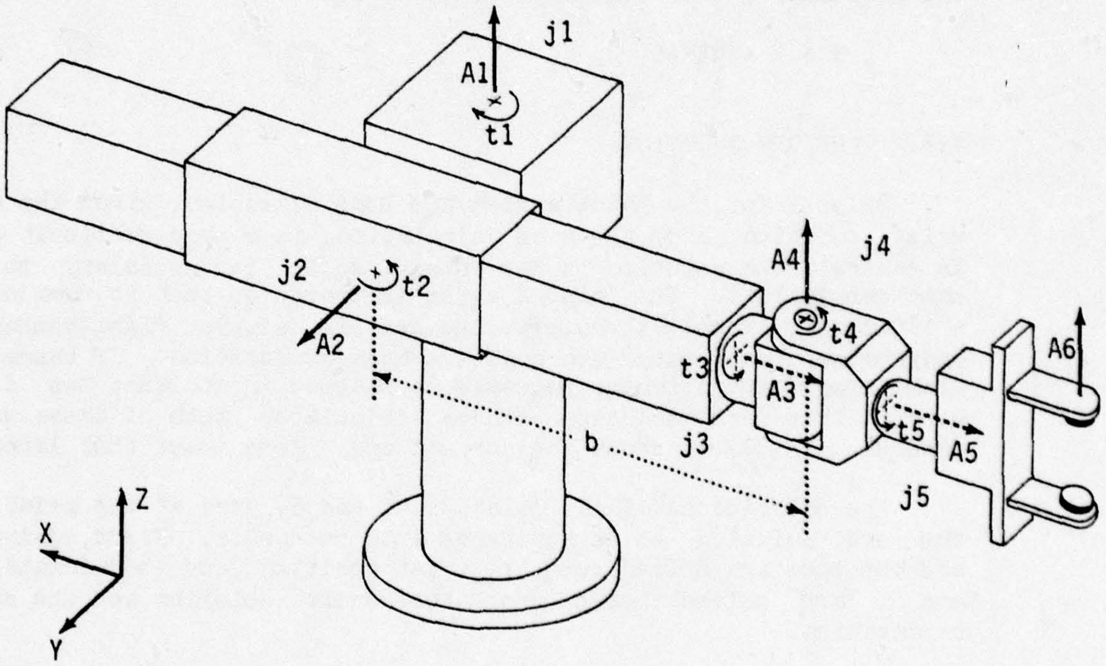

Figure 16 - Arm Solution Variables

Explicitly, the equations are:

```
A2 = A0*cos(t1) + A1 x A0*sin(t1)                 (A.2)
A3 = A1*cos(t2) + A2 x A1*sin(t2)                 (A.3)
A4 = A2*cos(t3) + A3 x A2*sin(t3)                 (A.4)
A5 = A3*cos(t4) + A4 x A3*sin(t4)                 (A.5)
A6 = A4*cos(t5) + A5 x A4*sin(t5)                 (A.6)
```

The position of the robot wrist is given by:

```
W = k1*A1 + k2*A2 + b*A3                          (A.7)
```

The position of the robot hand is given by:

```
H = W + k3*A5                                     (A.8)
```

A.2. THE ARM SOLUTION

Solving for the joint angles and boom extension, given the hand or wrist position and the hand orientation, is a more difficult problem. In general, the solution is not unique, so it is necessary to impose some constraints. The joint 2 angle is chosen so that it remains within a 180 degree range; therefore, the arm is always right-handed. The only other ambiguity concerns the hand orientation. If there were no limit stops, any orientation could be reached in at least two different ways. This arm solution scheme calculates both of those ways, and depends upon RSS to chose the correct one. More about that later.

The coincidence of the joint 3, 4, and 5, axes at the wrist allows the arm solution to be separated into two parts. First, joints 1, 2, and the boom are solved from the wrist position, and then joints 3, 4, and 5 are solved based upon the wrist solution and the specified orientation.

Position may be specified either as the hand position, or the wrist position. Orientation is specified by providing the hand orientation vectors A5, and A6. In RSS, R$FINGER corresponds to A5, and R$THUMB corresponds to A6.

If the hand position is given, the wrist position W is calculated using equation (A.8). Notice that from equation (A.7),

```
W - k1*A1 = k2*A2 + b*A3                          (A.9)
```

This is the equation of a right triangle with legs k2 and b, and hypotenuse |W - k1*A1|. Therefore, the boom extension is calculated by:

```
b = ( |W - k1*A1|**2 - k2**2 )**0.5               (A.10)
```

Taking the dot product of both sides of equation (A.9) with A1 yields:

```
(W - k1*A1).A1 = k2*A2.A1 + b*A3.A1               (A.11)
               = b*A3.A1
               = b*(A1.A1*cos(t2) + (A2 x A1).A1*sin(t2))
```

= b*cos(t2)

Therefore:

$$\cos(t2) = (W - k1*A1).A1/b \quad (A.12)$$

The value of t2 can be found from equation (A.12) using the right-handed arm constraint mentioned above.

Taking the dot product of both sides of equation (A.9) with A0 yields:

$$(W - k1*A1).A0 = b*\sin(t1)*\sin(t2) + k2*\cos(t1) \quad (A.13)$$

Taking the dot product of both sides of equation (A.9) with A1xA0 yields:

$$(W - k1*A1).(A1xA0) = -b*\cos(t1)*\sin(t2) + k2*\sin(t1) \quad (A.14)$$

At this point, all expressions except sin(t1) and cos(t1) are known. Equations (A.13) and (A.14) constitute two linear equations in these two unknowns. Because of the arm geometry, the solution simplifies to:

$$\cos(t1) = (k2*wy+b*\sin(t2)*wx)/(k2**2+(b*\sin(t2))**2) \quad (A.15)$$
$$\sin(t1) = (b*\sin(t2)*wy-k2*wx)/(k2**2+(b*\sin(t2))**2) \quad (A.16)$$

Once cos(t1) and sin(t1) are found, t1 is uniquely determined.

At this point, t1, t2, and b are known. Now, using the orientation specified by A5 and A6, the values of t3, t4, and t5 are determined.

Taking the dot product of both sides of equation (A.5) with A3 yields:

$$A5.A3 = \cos(t4) \quad (A.17)$$

Equation (A.17) determines the magnitude of t4, but not its sign. The remaining portion of the solution is carried out for both possible values of t4.

Taking the dot product of both sides of equation (A.5) with A2xA3 yields:

$$A5.(A2xA3) = \cos(t3)*\sin(t4) \quad (A.18)$$

Therefore:

$$\cos(t3) = A5.(A2xA3)/\sin(t4) \quad (A.19)$$

Taking the dot product of both sides of equation (A.5) with A2 yields:

$$A5.A2 = \sin(t3)*\sin(t4) \quad (A.20)$$

Therefore:

$$\sin(t3) = A5.A2/\sin(t4) \quad (A.21)$$

If t4 is not zero, equations (A.19) and (A.21) determine cos(t3) and sin(t3) which uniquely determine t3. If t4 is zero, it means that A3 and A5 are aligned, and the arm is in a degenerate position. The solution program substitutes the previous value of t3 and continues.

Taking the dot product of both sides of equation (A.6) with A4 yields:

$$A6.A4 = \cos(t5) \qquad (A.22)$$

Taking the dot product of both sides of equation (A.6) with A5xA4 yields:

$$A6.(A5xA4) = \sin(t5) \qquad (A.23)$$

Equations (A.22) and (A.23) determine cos(t5) and sin(t5) which uniquely determines t5.

The arm solution has now been found, except for the problems of a double-valued wrist solution, and position limits on the joints. The double-valued problem is solved by calculating the cost of moving the hand from its previous orientation to each of its two possible new orientations. The cost is nothing more than the sum of the angles spanned by joints 3, 4, and 5 as they move from their previous positions to their new positions. The orientation with the lowest cost is the one normally chosen.

All new joint values are checked against a table of limits which prevent the joint from striking its physical stops. If a joint would exceed its limit, it is forced back to the limit position, and flags are set indicating a positive or negative limit error for that joint. If the lowest cost wrist orientation causes a limit error, and the other orientation does not, RSS has the option of flipping the wrist, or saturating the joints as described above. If the hand is grasping something which cannot tolerate being turned over, or if the hand is working in close quarters, it is often better to have a slight orientation error than to make the extreme motion of flipping the wrist.

For the sake of posterity, the following list of A vector components is included:

```
A0      x = 0
        y = 1
        z = 0

A1      x = 0
        y = 0
        z = 1

A2      x = -sin(t1)
        y =  cos(t1)
        z =  0

A3      x =  cos(t1)*sin(t2)
        y =  sin(t1)*sin(t2)
        z =  cos(t2)

A4      x = -sin(t1)*cos(t3) - cos(t1)*cos(t2)*sin(t3)
        y =  cos(t1)*cos(t3) - sin(t1)*cos(t2)*sin(t3)
        z =  sin(t2)*sin(t3)
```

APPENDIX B

This appendix contains a description of the calculations performed by RSS to convert between joint torque (and boom force) values, and Cartesian force and torque vectors.

These calculations are based upon a static model of the robot arm. The forces and torques do not take into account the effects of acceleration, inertia, centrifugal or Coriolis forces. Therefore, the force applied is accurate only when the arm is stopped or moving slowly.

The A vectors, boom extension b, and arm constants k1, k2, and k3 are defined as in Appendix A, The Arm Solution. As before, lower case symbols represent scalars, upper case symbols represent three-element column vectors. Also, underlined variables represent 3x3 matrices. A superscript T indicates matrix transpose. Note that t is defined differently than in the arm solution.

t[n] denotes the magnitude of the torque at rotational joint n.
fb denotes the magnitude of the force at the boom, along A3.

B.1. WRIST FORCE TRANSFORMATION

Consider what happens when a force F is applied at the wrist. The torque T at each joint is given by the well-known formula:

$$T = R \times F \qquad (B.1)$$

where R is the moment arm about the joint. A particular joint is only sensitive to the component of T which is parallel to its axis.

Therefore:

$$\begin{aligned} t[n] &= A[n].T \\ &= A[n].(R[n] \times F) \end{aligned} \qquad (B.2)$$

Using a basic vector triple product identity yields:

$$t[n] = F.(A[n] \times R[n]) \qquad (B.3)$$

At joint 1:

$$R = b*A3 + k2*A2 \qquad (B.4)$$

And at joint 2:

$$R = b*A3 \qquad (B.5)$$

Substituting equations (B.4) and (B.5) into equation (B.3) yields:

$$t1 = F.(A1 \times (k2*A2 + b*A3)) \qquad (B.6)$$
$$t2 = F.(A2 \times (b*A3)) \qquad (B.7)$$

Since the boom is sensitive only to forces along its axis:

$$fb = F.A3 \tag{B.8}$$

Equations (B.6), (B.7), and (B.8), can be written in matrix form as:

$$\begin{vmatrix} t1 \\ t2 \\ fb \end{vmatrix} = \underline{FM} * F \tag{B.9}$$

where $\underline{FM}$ is the matrix:

$$\underline{FM} = \begin{vmatrix} (A1 \times (b*A3 + k2*A2))^T \\ (A2 \times (b*A3))^T \\ A3^T \end{vmatrix} \tag{B.10}$$

If F is known, the necessary joint torques to generate F are calculated using equation (B.9). If the joint torques have been estimated, the force at the wrist can be found using:

$$F = \underline{FM}^{-1} * \begin{vmatrix} t1 \\ t2 \\ fb \end{vmatrix} \tag{B.11}$$

If $\underline{FM}$ is singular, it means that the force due to joint 1 is parallel to the force due to either joint 2 or the boom. In this case, F is calculated by:

$$F = t1*k2*A3 \tag{B.12}$$

B.2. WRIST TORQUE TRANSFORMATION

The wrist torque is more easily obtained, since all the wrist joint axes intersect at the wrist. Consider applying a torque T to the wrist. Each joint is sensitive only to that component of the torque which is aligned with its axis.

Therefore:

$$t3 = A3.T \tag{B.13}$$
$$t4 = A4.T \tag{B.14}$$
$$t5 = A5.T \tag{B.15}$$

These equations can be combined into a single matrix equation:

$$\begin{vmatrix} t3 \\ t4 \\ t5 \end{vmatrix} = \underline{TM} * T \tag{B.16}$$

where $\underline{TM}$ is given by:

$$\underline{TM} = \begin{vmatrix} A3^T \\ A4^T \\ A5^T \end{vmatrix} \tag{B.17}$$

If T is known the joints torques which generate T are found by using equation (B.16). If the joint torques are known, the total torque at the wrist if given by:

$$T = \underline{TM}^{-1} * \begin{vmatrix} t3 \\ t4 \\ t5 \end{vmatrix} \qquad (B.18)$$

If $\underline{TM}$ is singular, it means that the joint 3 axis is aligned with the joint 5 axis. In this case, T may be calculated by:

$$T = t4*A4 + \min(t3,t5)*A3 \qquad (B.19)$$

APPENDIX C

This appendix explains the stereo vision hardware used by RSS, the method used to transform between robot and vision coordinates, and the vision system calibration procedure.

The vision hardware consists of a single vidicon TV camera interfaced to a PDP10 computer [27]. Stereo information is obtained by using the mirror arrangement shown in Figure 15. These mirrors convert the single TV image into two images which appear to come from two virtual cameras. The relative orientation of these virtual cameras is adjusted by rotating the two outside mirrors. There is no fundamental reason why two real TV cameras could not replace the single camera and the mirrors.

Each image is considered to lie on a plane which is normal to its virtual camera direction, at a distance of f from the position of the virtual camera. The main image coordinate system has its origin at the lower left hand corner. Increasing x corresponds to moving from left to right. Increasing y corresponds to moving from bottom to top. The left image coordinates range from 0 to 116, the right image coordinates range from 136 to 237, which leaves a 20 pixel dead zone for the center mirror corner. The lens center is at the point (126,119).

C.1. ROBOT TO VISION COORDINATE TRANSFORMATION

Robot coordinates are transformed to vision coordinates as follows: Let P be an arbitrary point in robot space. The position of P with respect to the camera position C is calculated, and the result is rotated using the matrix VRCT to yield a vector V.

$$V = \begin{vmatrix} xv \\ yv \\ zv \end{vmatrix} = \underline{VROT} * (P - C) \tag{C.1}$$

The distance from C to the plane containing P, normal to the camera direction, is given by zv. To compensate for perspective, V is scaled so that its third component has the value f. The first two components are then the image coordinates relative to the lens center, which must be translated to yield the actual image coordinates (x,y).

$$x = xv * \frac{f}{zv} + xc \tag{C.2}$$

$$y = yv * \frac{f}{zv} + yc \tag{C.3}$$

where xc = 126, and yc = 119. This procedure must be performed once for each of the two virtual cameras, using the appropriate C, VROT, and f.

C.2. VISION TO ROBOT COORDINATE TRANSFORMATION

Given the vision coordinates (x,y) for both the right and left images, the corresponding point P in robot coordinates is found as follows: The point in vision space is expressed relative to the virtual camera position at (xc, yc, f), and rotated using the matrix <u>RROT</u> to yield the vector V.

$$V = \underline{RROT} * \begin{vmatrix} x - xc \\ y - yc \\ f \end{vmatrix} \qquad (C.4)$$

The direction of V is then calculated as the vector U:

$$U = V/|V| \qquad (C.5)$$

This procedure is performed twice to calculate a value of U for both the right and left image coordinates. Let these two U vectors be denoted UR and UL, which result from the images of the virtual cameras at CR and CL respectively.

UR and UL are two direction vectors from two points in robot space. If everything were perfect, one could simply solve for the intersection of these two rays. However, it is better to find the midpoint of the line segment which connects the two rays at their point of closest approach, as recommended by Duda and Hart [29]. The value of P is calculated by:

$$P = \frac{(UR.D-UR.UL*UL.D)*UR+(UR.UL*UR.D-UL.D)*UL}{2*(1 - (UR.UL)**2)} + \frac{CL+CR}{2} \qquad (C.6)$$

where D = CL - CR

C.3. VISION SYSTEM CALIBRATION PROCEDURE

This section discusses the calibration procedure which determines C, f, <u>VROT</u>, and <u>RROT</u>. During calibration, a small light is attached to the robot hand. The robot is moved manually to two widely spaced points S1 and S2 in the field of view. The robot then moves to eight positions, while the calibration program records the robot positions and the corresponding light position as seen by both virtual cameras. The robot position is stored as a 3-vector in robot coordinates, and the light position is stored as a 3-element vector containing the right image x coordinate, the left image x coordinate, and the average of the two y coordinates. The eight robot positions are denoted P1 through P8, and the corresponding image coordinates are denoted I1 through I8. The robot positions are listed below:

```
P1 = S1
P2 = S1 + k*X
P3 = S1 + k*Y
P4 = S1 + k*Z
P5 = S2
P6 = S2 + k*X
P7 = S2 + k*Y
```

$P8 = S2 + k*Z$

where k is some constant much smaller than the distance from S1 to S2, and X, Y, and Z are the unit vectors pointing along the robot x, y, and z axes. Two sets of matrices are defined as follows:

$$\underline{PM1} = (\ P2-P1\quad P3-P1\quad P4-P1\ )$$
$$\underline{VM1} = (\ V2-V1\quad V3-V1\quad V4-V1\ )$$

$$\underline{PM2} = (\ P6-P5\quad P7-P5\quad P8-P5\ )$$
$$\underline{VM2} = (\ V6-V5\quad V7-V5\quad V8-V5\ )$$

To find C, calculate the matrices $\underline{L1}$ and $\underline{L2}$, which relate small motions in image coordinates to small motions in robot coordinates, about the points P1 and P5 respectively.

$$\underline{L1} = \underline{PM1} * \underline{VM1}^{-1} \qquad (C.7)$$
$$\underline{L2} = \underline{PM2} * \underline{VM1}^{-1} \qquad (C.8)$$

If k is sufficiently small, the errors in direction due to perspective effects will also be small. Let T be the vector $(0,1,0)^T$. If T is interpreted as an image vector, it represents a small motion to the right in the left image, and no motion in the right image. Such motion must be directed toward the right virtual camera. Therefore:

$$U1 = \underline{L1} * T \qquad (C.9)$$
$$U2 = \underline{L2} * T \qquad (C.10)$$

are vectors whose directions are toward the right camera, from the points P1 and P5. The position of the right virtual camera is found using this information and a formula similar to equation (C.6). The position of the left virtual camera is found in a similar manner, using $T = (-1,0,0)^T$.

Now that the values of C for the right and left cameras have been found, the value of f, the distance from the image plane to the camera position is calculated. Let U1 and U2 be vectors from a single camera position to two distinct points in robot space. If the points are in view, there will also be two vectors V1 and V2 which point from the camera position to the points where U1 and U2 intersect the image plane. Since V1 and V2 have the same directions as U1 and U2, the angle between each pair of vectors must be the same. The x and y coordinates of V1 and V2 are the positions of the lights in the image plane. The z coordinate is f, and must be found. The problem may be restated as follows: Given two direction vectors U1 and U2, and four scalars x1, y1, x2, and y2, find f such that the angle between the vectors (x1,y1,f) and (x2,y2,f) is equal to the angle between U1 and U2. Since we are not interested in the sign of the angle, and the geometry of the vision hardware restricts the angle to values less than 90 degrees, it is reasonable to use the cosine of the angles rather than the angle itself. Let t represent the cosine of the angle between the vectors U1 and U2.

$$t = U1.U2 \qquad (C.11)$$

The problem is to find an f such that:

$$t = \frac{x1*x2+y1*y2+f**2}{[(x1**2+y1**2+f**2)*(x2**2+y2**2+f**2)]**0.5} \qquad (C.12)$$

Let: a represent x1*x2 + y1*y2
b represent x1**2 + y1**2
c represent x2**2 + y2**2

Substituting into equation (C.12) yields:

$$t = \frac{a+f**2}{[(b+f**2)*(c+f**2)]**0.5} \qquad (C.13)$$

Squaring each side yields:

$$t**2 = \frac{a**2+2*a*f**2+f**4}{(b+f**2)*(c+f**2)} \qquad (C.14)$$

$$= \frac{a**2+2*a*f**2+f**4}{b*c+(b+c)*f**2+f**4}$$

Collecting and rearranging terms yields:

$$(t**2-1)*f**4+(t**2*(b+c)-2*a)*f**2+b*c*t**2-a**2 = 0 \qquad (C.15)$$

which is quadratic in f**2. The value of f**2 is found numerically, using the quadratic formula. The value of f is then calculated as -(f**2)**0.5 since the camera position is behind the image plane.

This procedure is performed once for each virtual camera, using P1 and P5 to find U1 and U2, and substituting the appropriate image coordinates for x1, y1, x2, and y2 from I1 and I2.

Now the rotation matrices which transform vision coordinates to robot coordinates are calculated. First, a number of vectors are formed:

$$V1 = \begin{vmatrix} I1 \text{ right } x \\ I1\ y \\ f \end{vmatrix}$$

$$V2 = \begin{vmatrix} I5 \text{ right } x \\ I5\ y \\ f \end{vmatrix}$$

V3 = V1 x V2

U1 = P1 - C
U2 = P5 - C
U3 = U1 x U2

These vectors are combined into matrices:

$$\underline{VMAT} = \left| \begin{array}{ccc} \frac{V1}{|V1|} & \frac{V2}{|V2|} & \frac{V3}{|V3|} \end{array} \right|$$

$$\underline{UMAT} = \left| \begin{array}{ccc} \frac{U1}{|U1|} & \frac{U2}{|U2|} & \frac{U3}{|U3|} \end{array} \right|$$

Finally, the rotation matrices are calculated by:

$$\underline{RROT} = \underline{UMAT} * \underline{VMAT}^{-1} \qquad (C.16)$$

$$\underline{VROT} = \underline{RROT}^{-1} \qquad (C.17)$$

This procedure is also performed for the left camera, substituting the appropriate values for I1 x, f, and C.

APPENDIX D

This appendix describes the servoing process from the individual joint's point of view. It includes the joint model used for servo calculations and external torque estimates.

Each manipulator joint is driven by a permanent magnet DC motor which is interfaced to a PDP11 minicomputer via a hardware controller. This controller determines the motor current based upon the output of a D/A converter and feedback from a tachometer. The angular position and velocity of each joint are also available to the computer via A/D converters.

A block diagram of an entire joint servo is shown in Figure 17. The servo calculation routine is executed 15 times per second. It reads the robot joint positions and velocities, and calculates the D/A value, taking into account the desired angular position, the desired joint torque, the gravitational loading of the joint, and the limitations of the joint motors. This procedure is discussed more fully in section 5.3. Although the system appears to be straightforward, it defies analysis for the following reasons:

1. The robot inertia J_r is not constant, but depends upon the positions of all the robot joints.

2. The robot friction F is large and appears to vary with joint position and orientation.

3. The algorithm used to calculate the D/A value is extremely complicated and difficult to model.

For these reasons, an empirical approach was taken to determining the joint servo parameters.

The value of K_h is constrained by the physical limits of the joint motors. The hardware controller is set to achieve the maximum allowable motor current when the D/A is at its maximum value. The parameters K_m, K_r, F, J_r and B_r are determined by the robot manipulator mechanism. The effect of motor inductance is negligible compared with the effect of joint inertia. The hardware velocity feedback gain B_h was chosen empirically for each joint to eliminate overshoot during step position motions with average inertial loading. Because of the low sampling rate, B_h values are large and cause the robot to move slowly.

The external force estimator also uses the joint model to compute the external torque at each joint. The method used is described below:

Consider the transfer function between the D/A converter and the joint velocity state variable. A signal flow graph of this portion of the controller is shown in Figure 18.

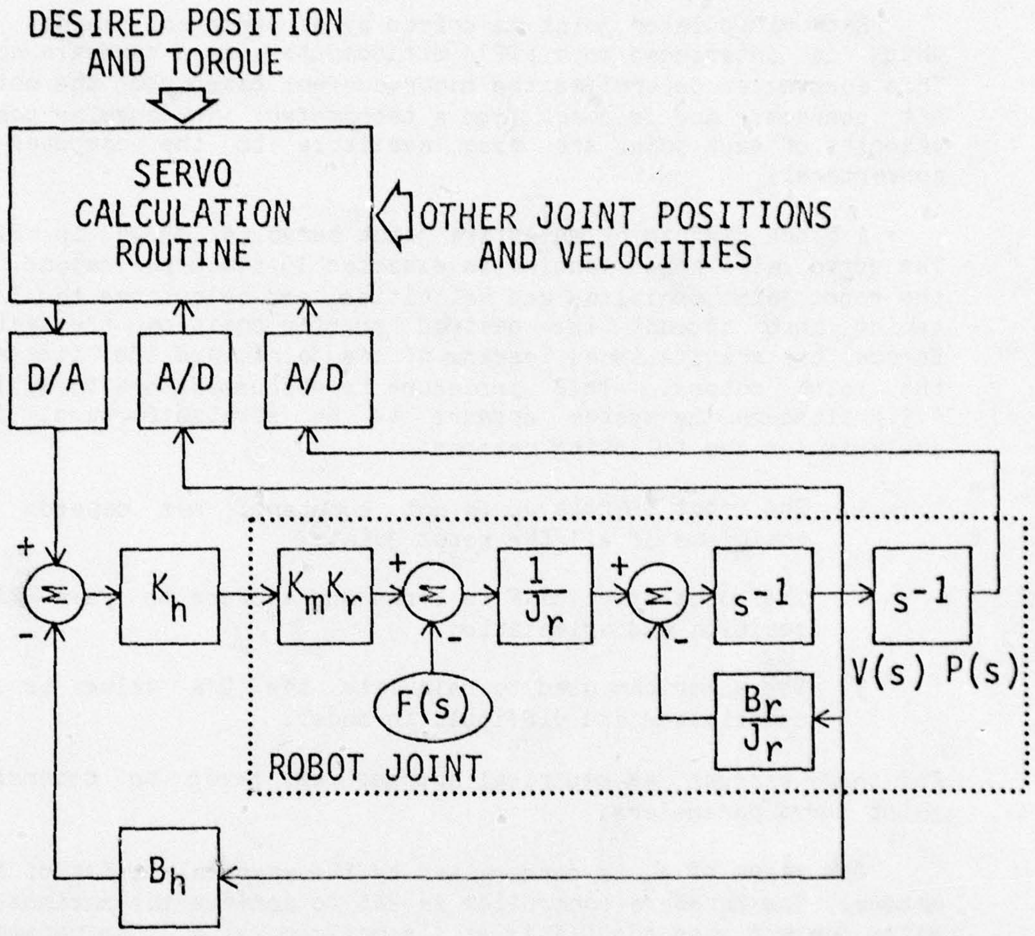

Figure 17 - Joint Servo Block Diagram

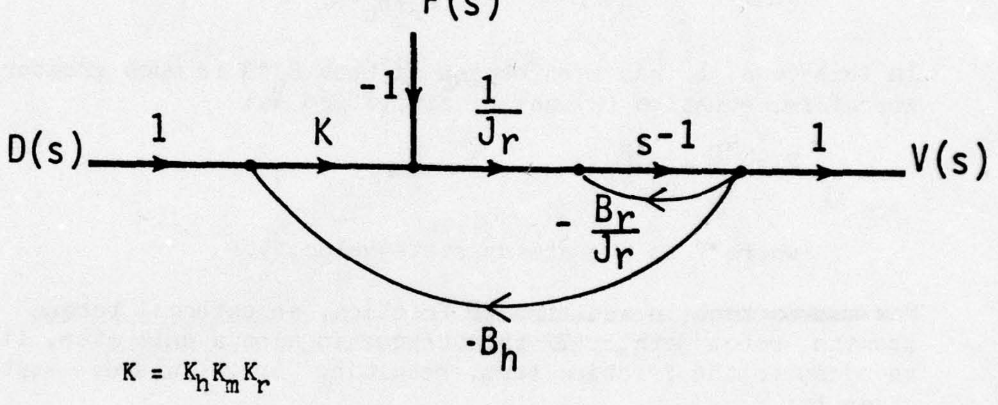

Figure 18 - D/A to Velocity Transfer Function

The transfer function is found to be:

$$V(s) = \frac{K*D(s)-F(s)}{J_r*s+B_r+B_h*K} \quad (D.1)$$

where V(s) is the robot joint velocity
D(s) is the D/A output
F(s) is the robot friction

If the D/A output and robot friction are assumed to be step functions with magnitude D and F respectively, applying the final value theorem yields:

$$\lim_{t\to\infty} V(t) = \lim_{s\to o} s*V(s) = \frac{K*D-F}{B_r+B_h*K} \quad (D.2)$$

In this case, B_h has been chosen so that B_h*K is much greater than B_r; therefore, equation (2) may be simplified as:

$$V = \frac{D}{B_h} - \frac{F}{B_h*K} \quad (D.3)$$

where V is the steady state velocity.

Now assume that in addition to friction, an external torque is applied to the robot joint. If this torque is also a unit step, it can simply be added to the friction term, resulting in a steady state velocity given by:

$$V = \frac{D}{B_h} - \frac{T+F}{B_h*K} \quad (D.4)$$

Solving for T+F yields:

$$T+F = K*(D-B_h*V) \quad (D.5)$$

Since D and V are known by the computer, finding K and B_h will allow T+F to be calculated. The external force estimator includes friction as an external force, so the joint torque calculated really is T+F. It should be noted that the external torques calculated are valid only when the manipulator joints are moving with constant velocity.

The value of B_h was determined from equation (D.3) by plotting the steady state joint velocity V as a function of the D/A value D. The reciprocal of the slope is B_h. The value of K was determined by plotting the static joint torque as a function of D/A value, obtained while holding the joint at a fixed position. The slope of this plot is K.

APPENDIX E

This appendix is a user's manual for RSS. The first part describes commands for establishing robot control servos, the second part describes RSS program commands, the third part describes commands to create, execute, edit, and save RSS programs, and the last part describes the program editor.

E.1. SERVO COMMANDS

Servo commands allow robot servos to be declared, modified, and examined. These commands may be typed directly to RSS for immediate execution, or they may be entered into an RSS program for execution later. Even while a program is running, the user may enter immediate commands which alter some servo declaration or function definition. Parameters may be passed to a program by defining functions before starting the program, and referencing those functions within the program.

All robot activity is controlled by declaring software servos which control the robot's position, orientation, force, and torque. These declarations use vector and scalar expressions which may use any system-defined or user-defined functions. If an active servo attempts to reference an undefined function, the robot will execute a panic stop and wait for the error to be corrected.

E.1.1 VECTORS AND SCALARS

RSS deals with 3-element vectors and scalars. The vectors may represent a position or direction in robot coordinates. The robot coordinate system is Cartesian with its origin at the center of the robot base, at table level. The three components of a robot position, denoted x, y, and z, correspond to space as follows: x corresponds to distance along the length of the robot table, y corresponds to distance along the breadth of the robot table, and z corresponds to distance normal to the plane of the robot table. The units for position components are centimeters. Direction vectors have unit magnitude and represent a direction in the robot coordinate system.

E.1.2 VECTOR AND SCALAR EXPRESSIONS

Servo declarations and function definitions usually refer to a vector or scalar expression. These expressions consist of vector or scalar constants or functions combined with vector or scalar operators. Scalar function names must begin with a lower case alphabetic character, vector function names must begin with an upper case alphabetic character. Names may also contain digits and dollar signs. Only the first eight characters of a name are significant.

The scalar operators are:

+ Scalar addition, the operands are scalars, the result is a scalar.
- Scalar subtraction or negation, the operands are scalars, the result is a scalar.
* Scalar multiplication, the first operand must be a scalar, the result is of the same type as the second operand.
/ Scalar division, the second operand must be a scalar, the result is of the same type as the first operand.

The vector operators are:

+ Vector addition, the operands are vectors, the result is a vector.
- Vector subtraction or negation, the operands are vectors, the result is a vector.
. Vector dot product, the operands are vectors, the result is a scalar.
Vector cross product, the operands are vectors, the result is a vector.

In general, addition and subtraction have lower precedence than the other operators. The vector dot product has lower precedence than the vector cross product. When precedence is equal, an attempt is made to evaluate expressions so that the operand types are appropriate for the operators encountered. For example, in the expression:

```
A#b*C
```

the vector cross operator (#) sees a vector operand A, and a scalar operand b. The cross operation will be deferred in the hope that continued evaluation to the right will eventually result in a vector value for the second operand. In ambiguous situations, programmers are urged to use parentheses.

There are also some special functions:

[s1,s2,s3]	Returns a vector value which has as its components, the three scalar expressions s1, s2, and s3.
<VEXP>	Returns a vector value equal to the vector expression VEXP normalized to unit magnitude. Also known as the direction of VEXP.
|VEXP|	Returns a scalar value equal to the magnitude of the vector expression VEXP.
|sexp|	Returns the absolute value of the scalar expression sexp.
{exp}	Returns the value of any expression as a constant. When this function appears in a servo command or a program statement, the expression is immediately evaluated and made a constant. The expression may be either a vector expression or a scalar expression. Note: when used in a program, evaluation takes place when the program statement is executed, not when the statement is inserted into the program.

E.1.3 SYSTEM-DEFINED CONSTANTS

For convenience, the following constants are already defined in RSS and may be referenced in any expression.

B$X - A unit vector in the X direction, equal to [1,0,0].

B$Y - A unit vector in the Y direction, equal to [0,1,0].

B$Z - A unit vector in the Z direction, equal to [0,0,1].

ZERO - The zero vector, equal to [0,0,0].

E.1.4 SYSTEM-DEFINED FUNCTIONS

The following functions are already defined by RSS and may be referenced in any expression. The values of these functions are constantly updated at interrupt level to reflect the state of the robot at the last sampling instant. Figure 3 illustrates some of these functions.

R$GRIP - A vector indicating the robot gripper position expressed in robot coordinates. The units are centimeters.

R$WRIST - A vector indicating the robot wrist position expressed in robot coordinates. The units are in centimeters.

R$FINGER - A direction vector indicating the direction from the robot wrist to the robot fingers.

R$THUMB - A direction vector indicating the direction from robot finger two to robot finger one. The touch sensor is on finger one.

R$FORCE - A vector indicating the estimated external force on the robot wrist. The direction of this vector is the direction in which the force is being applied, and the magnitude is the magnitude of the force, in newtons.

R$TORQUE - A vector indicating the estimated external torque on the robot wrist. The direction of this vector is the axis about which the torque is being applied, and the magnitude is the magnitude of the torque, in newton-meters.

r$hand - A scalar containing the distance between the robot fingers. Units are centimeters.

r$touch - A scalar containing the finger 1 touch sensor reading. Units are undefined, but 20 is about 2 ounces, and 220 is about 5 pounds.

E.1.5 USER-DEFINED FUNCTIONS

To improve program readability, and to avoid recomputing common subexpressions in servo declarations, the programmer may define his own functions. The rules for function names listed above apply to user functions. The expression defining a function must evaluate to the same type (vector or scalar) as the function name. System defined constants and functions may not be redefined. A user may freely redefine a function used in a servo declaration while the servo is active. In fact, such redefinition is a convenient method of dynamically altering robot action. The syntax for function definition is shown below:

define VNAME = VEXP

where VNAME is a 3-vector function name, and VEXP is any vector expression.

define sname = sexp

where sname is a scalar function name, and sexp is any scalar expression.

E.1.6 CLOCK DECLARATIONS

The clock declaration allows programmers to define functions which vary with time. The syntax for a clock declaration is given below:

clock sname=sexp

where 'sname' is the name of a scalar function, and 'sexp' is any scalar expression.

The scalar expression is evaluated and used as the intial setting of the clock. Any reference to the function 'sname' will return the initial setting plus the number of seconds which have elapsed since the declaration was made. The clock value is updated 15 times per second. To reset a clock function, it is merely redeclared. Clock functions may not be redefined to be non-clock functions, neither may non-clock functions be redefined to be clock functions.

E.1.7 FORCE AND TORQUE SERVO DECLARATIONS

These declarations establish open loop servos which exert a force or torque at the robot wrist. Force servos are considered secondary to position servos; therefore, if there is a conflict between the force servo and the position servo, the force servo will be modified. Also, position declarations will augment the force servo to counteract any external forces which would cause an error in position. Torque servos are considered secondary to orientation servos; therefore, if there is a conflict between the torque servo and the orientation servo, the torque servo will be modified. Also, orientation declarations will augment the torque servo to counteract any external torques which would cause an error in orientation.

force VEXP

Establish a servo which exerts a force at the robot wrist. The direction of the force is given by the direction of VEXP, and the magnitude of the force is given by the magnitude of VEXP. The units are newtons.

torque VEXP
Establish a servo which exerts a torque at the robot wrist. The direction of the torque axis is given by the direction of VEXP, and the magnitude of the torque is given by the magnitude of VEXP. The units are newton-meters.

E.1.8 POSITION SERVO DECLARATIONS

RSS controls the robot's position by servoing the wrist to move a reference point to some position goal. When a position servo is active, the robot is assumed to have control over the reference point, which is determined by the expression to the left of the equal sign in the declaration.

position none
Cancel any position servos now active, and allow the robot to move freely in compliance with any external force or force servo. Leave any declared force servos unmodified.

position point REF = POINT
Establish a servo which moves the wrist so that the point determined by REF moves to the point determined by POINT. Both REF and POINT may be arbitrary vector expressions. Ignore any declared force servo and substitute a force servo which opposes any external force on the wrist.

position line REF = POINT;DIR
Establish a servo which moves the wrist so that the point determined by REF moves to the line determined by the point POINT and the direction DIR. Modify the declared force servo to ignore any force components normal to DIR, and to counteract any external forces normal to DIR. REF, POINT, and DIR may be arbitrary vector expressions.

position plane REF = POINT;NORMAL
Establish a servo which moves the wrist so that the point determined by REF moves to the plane determined by the point POINT and the normal to the plane NORMAL. Modify any declared force servo to ignore any force components in the direction of NORMAL, and to counteract any external forces in the direction of NORMAL. REF, POINT, and NORMAL may be arbitrary vector expressions.

E.1.9 ORIENTATION SERVO DECLARATIONS

RSS controls the robot's hand orientation by rotating the hand about the wrist to align reference direction vectors with some goal direction vectors. When an orientation servo is active, the robot is assumed to have control over the reference vector or vector pair, which is determined by the expression or expression pair to the left of the equal sign in the declaration.

orient none
Cancel any orientation servos now active, and allow the robot hand to change orientation freely in compliance with any external torque or torque servo. Leave any declared torque servo unmodified.

orient align REF = DIR
Establish a servo to rotate the wrist so that the direction vector determined by REF is aligned with DIR. Modify the declared torque servo to ignore any torque component normal to DIR, and to counteract any external torque normal to DIR. REF and DIR may be arbitrary vector expressions. Only the directions of these expressions are important.

orient fixed REF1;REF2 = DIR1;DIR2
Establish a servo to rotate the wrist so that the pair of direction vectors determined by REF1 and REF2 is aligned with the pair of vectors determined by DIR1 and DIR2. Ignore any declared torque, substituting a servo which counteracts any external torque. REF1, REF2, DIR1, and DIR2 may be arbitrary vector expressions. Only the directions of these expressions are important. Note: Orient can deal with degenerate reference specifications.

E.1.10 PRIMITIVE SERVO DECLARATIONS

The following primitive servo declarations provide a means of circumventing the software which modifies force and torque servos due to position or orientation declarations.

position goal REF = POINT
Establish a servo which moves the wrist so that the point determined by REF moves to the point determined by POINT. Both REF and POINT may be arbitrary vector expressions. Do not modify the declared force servo.

orient goal REF1;REF2 = DIR1;DIR2
Establish a servo to rotate the wrist so that the pair of direction vectors determined by REF1 and REF2 is aligned with the pair of vectors determined by DIR1 and DIR2. Do not modify the declared torque servo.

E.1.11 HAND CONTROL COMMANDS

The operation of the robot hand is totally independent of the other servoing operations, and is controlled by the following commands:

hand none
Cancel any hand servo and release the finger brakes.

hand brake
Cancel any hand servo and set the finger brakes.

hand position sexp
Establish a servo which moves the fingers so that the distance between them is 'sexp'. Units are in centimeters.

hand squeeze sexp

Establish a servo which uses feedback from the finger pressure sensor to move the fingers together with a force of 'sexp'. A value of 20 corresponds to about 2 ounces, and a value of 220 corresponds to about 5 pounds.

E.1.12 SERVO CONTROL COMMANDS

These commands allow the entire software servo to be started and stopped so that a number of servo declarations can be made and then started simultaneously. Also, the programmer can temporarily suspend and continue the robot servo during debugging.

servo on

Activate the software servo so that the robot will move in accordance with any declared servos. This command will restart the robot servo after is has been suspended by a 'servo off' command, by a program 'stop' command, or by pressing the panic button. Note that RSS initially starts with the servo suspended.

servo off

Suspend the software servo and stop the manipulator. This command may be cancelled via a 'servo on' command with no effect upon declared servos.

E.1.13 FUNCTION OUTPUT

The following commands allow the user to examine any user-defined or system-defined functions.

list fname

List the definition for the function named 'fname' at the user terminal. If 'fname' is null, all user-defined function definitions are listed.

type exp

Type the value of the expression 'exp' at the user terminal. The expression may evaluate to either a scalar or vector value.

E.1.14 MISCELLANEOUS COMMANDS

clear

Clears the valid bit for all function values so that function references cause the function to be evaluated instead of returning the last value. This command is useful only for debugging when the robot is disabled by the initial dialog. Normally, valid bits are cleared at interrupt level by the robot servo.

scale sexp

Sets the servo scale factor to the value given by the scalar expression 'sexp'. This scale factor is part of the positional error gain used by the robot servo. Initially, the scale factor is set to 100.

E.1.15 VISION PROCESSOR COMMANDS

These commands send messages to the vision processor software which currently runs on a PDP10 computer. The vision functions declared may be used in expressions and declarations just like any other user or system-defined functions. Like the system-defined functions, their values are automatically updated at interrupt level. Vision functions may not be redefined as non-vision functions, and non-vision functions may not be redefined as vision functions.

vision VNAME
Declare to RSS that vector function VNAME is defined by the vision processor. Any servo which uses VNAME will be dependent upon visual data.

locate VNAME
Instruct the vision processor to locate VNAME once and return its value. The entire image is searched.

locate VNAME at POSITION
Instruct the vision processor to locate VNAME once and return its value. The image is searched by spiraling outward from the robot coordinates given by the 3-vector POSITION.

track VNAME
Instruct the vision processor to continually track the position of VNAME, and continually inform RSS of its current position.

forget VNAME
Instruct the vision processor to stop tracking VNAME.

E.2. RSS PROGRAM STATEMENTS

The statements described in this section are only recognized when executed as part of a program. Additionally, any servo command may be used as program statement.

Any program statement which has a semicolon (;) in the first character position is a comment and ignored during execution.

Any program statement except a comment may have a label. Labels follow the normal rules for RSS function names: they must be alphanumeric and begin with an alphabetic character, and the maximum label length is 8. Labels must be followed immediately by a colon (:).

Logical expressions have a value of either true or false as determined by evaluating an arithmetic relation or by testing a specified flag. The syntax for arithmetic relations is:

```
sexp; relation sexp
```

where 'sexp' is any scalar expression, and 'relation' is one of the following keywords:

```
lss - less than
leq - less than or equal to
eql - equal to
geq - greater than or equal to
gtr - greater than
neq - not equal to
```

The syntax for flag testing is:

```
flag flagtype fname
```

where 'fname' is the name of a function. Currently, the only recognized flagtype keywords are 'lost' and 'found' which, when used to test a vision function, indicate whether the feature associated with that function is lost or found. Vision functions are not flagged lost or found until the vision processor has actually attempted to locate or track the associated feature.

Some examples of logical expressions are:

```
|error|; lss 0.5
flag lost BOLT
```

The first expression is true if the absolute value of the scalar function 'error' is less than 0.5, and the second expression is true if the feature associated with the vision function 'BOLT' has been lost.

Below is a list of the various program statements with a description of their effect:

goto label
: Unconditionally branch to the statement labeled 'label'. Also check any condition monitors declared by a trap statement.

if lexp; statement
: Evaluate the logical expression 'lexp' and if it is true, execute the statement following the semicolon (;). Otherwise continue execution with the statement on the following line. The specified statement may be any program statement or servo command.

pause
: Suspend program execution until the word 'continue' is typed at the user terminal. Even though the RSS program is suspended, any declared servos remain active and the robot continues to move.

print string
: Print the specified string at the user terminal. If the last character of the string was a semicolon (;), suppress it as well as any trailing carriage return.

stop
: Stop the robot program and disable all condition monitors, but leave the servos active. Note that the effect of a stop executed from a program is different from a stop typed directly to RSS.

trap prio to label on lexp;

Establish a condition monitor with priority 'prio' which will asynchronously jump to the statement labeled 'label' if the logical expression 'lexp' ever is true. Trap priorities range from 0 to 255, with 0 being the lowest priority. If more than one trap fires simultaneously, the one with the highest priority is recognized. In the event of a tie, the most recently declared trap is recognized. If a trap fires, it disables all other traps with priorities less than or equal to its own, but does not affect traps with higher priority. To prevent interrupting a multiple statement servo declaration, traps are evaluated only while the program is in wait state, pause state, or when a goto statement is executed.

untrap prio

Disable all traps with priority less than or equal to 'prio'. If 'prio' is null, disable all traps.

wait until lexp;

Suspend program execution until the specified logical expression 'lexp' is true, and then continue program execution with the next statement. Program execution may also resume if a trap fires. A wait statement is actually a priority 0 trap to the following statement. Even though program execution is suspended, any declared servos remain active and the robot continues to move.

wait while lexp;

Suspend program execution until the specified logical expression 'lexp' is false. Otherwise this statement is the same as the 'wait until' statement described above.

E.3. TOP LEVEL COMMANDS

The commands described in this section allow the programmer to create, execute, edit, save, and restore RSS programs.

RSS commands may be read from a disk file instead of the terminal by using the command:

```
require filename
```

which directs RSS to read from the file named 'filename' until the end of the file. Require commands may be nested to a depth of 14. Require commands provide a method of quickly defining frequently used functions, and restoring RSS programs which have been stored via the 'save' command.

RSS program names follow the same convention used for naming functions: the name must be alphanumeric, it must begin with an alphabetic character, and the maximum length is 8.

The following is a list of RSS top level commands with a description of their effect:

continue
: Resume executing any RSS program which has been suspended by a 'pause' statement.

edit prog
: Enter the program editor to modify the program named 'prog'. If 'prog' is null, edit the last program referenced by an edit, make, or run command. A program which is currently running may not be edited.

make prog
: Create a program named 'prog' and enter the editor in insert mode.

run prog
: Begin execution of the program named 'prog'. If 'prog' is null, execute the program last referenced by an edit, make, or run command. Only one program may be running at a time.

save prog filename
: Save the program named 'prog' in a disk file named 'filename'. If 'filename' is null, save the program in file called 'prog.ROB'. This command creates a file which can be restored by the require command. It does so by inserting the line 'MAKE prog' at the beginning of the file, and <esc><esc> at the end.

E.4 THE RSS PROGRAM EDITOR

This editor allows RSS programs to be created or modified without exiting from RSS. Editing commands allow the user to position a line pointer, and to insert, delete, or display text at the current pointer location. The following is a list of the editing commands and their function:

i - Enter insert mode. Any characters which follow the 'i' are inserted into the text, before the line pointer, until an <esc> character is typed.

j - Move the line pointer to the first line of the program.

k - Delete the current line and type the next line.

l - Move the line pointer forward one line and type the current line.

r - Move the line pointer backward one line and type the current line.

t - Type the current line.

z - Move the line pointer to past the last line.

<tab> - Equivalent to typing 'i<tab>'. Enter insert mode after inserting a tab.

<esc> - If in insert mode, exit to editor command level. If at editor command level, exit to RSS command level.

The 'k', 'l', and 'r' commands may be preceded by a repeat count which consists of a decimal number or the letter 'h'. The 'h' indicates that the command is to be repeated until the end of the program is reached.

APPENDIX F

This appendix provides a detailed description of the RSS software. Almost all of RSS has been written in MACRO-11 to squeeze every bit of speed out of the PDP11/40. The remaining routines were written in BLIS11.

RSS may be divided into three main sections:

1. Servo declaration and control.
2. Interrupt level servo calculations.
3. RSS program editing and execution.

Each of these sections contains one or more of the blocks shown in Figure 13.

F.1 SERVO DECLARATION AND CONTROL

This section contains the Servo Command Processor, the User Functions, the System Functions, and the Servo Declarations.

INTERP

This module contains routines for defining user functions, evaluating those functions, declaring robot servos, evaluating the error terms for those servos, and controlling the sampled data servo.

User and system functions are represented as lists of atoms which contain: the function name, a pointer to the function value, a pointer to the routine or interpreter code necessary to evaluate the function, and flags which indicate the function type (scalar or vector), its validity, and other information. Vector components and scalars are single-precision PDP11 floating point numbers, stored with the low order word first (lowest address) followed by the high order word (which contains the exponent). A pointer to a scalar value points to the low order word, a pointer to a vector value points to the low order word of the first component in a six-word block. Vectors are stored as (from low address to high): X low, X high, Y low, Y high, Z low, Z high. Interpreter values are passed in the Vector ACcumulator (VAC), a seven word block which contains a value-type word followed by a vector value. Scalars are passed as the X component of the vector.

Below is a brief description of the main routines in INTERP:

ENCODE(stringptr) - Reads an ASCIZ string at stringptr which should contain a scalar or vector expression. Translates that string into tokens which can be evaluated efficiently at a later time.

DECODE(codeptr) - Reads the tokenized code at codeptr and translates it back into an ASCIZ string for output.

EVAL - Evaluates tokenized code and returns the value in VAC.

GETVAL(fnptr) - Returns the value of the function at fnptr in VAC. If the value of the function is already valid, it is simply copied to the VAC, otherwise, the function is evaluated first. Functions may be defined in terms of interpreter code which must be evaluated by EVAL, or in terms of an executable subroutine.

EXECUTE(stringptr) - Reads the ASCIZ string at stringptr which contains a servo command. These commands are described in Appendix E. This routine is the interface between the entire servo command section and the user commands or programs.

User function definitions consist of creating a function definition atom with the appropriate flags and name, ENCODE'ing the defining expression, and storing a pointer to the tokenized code in the atom. Servo declaration consists of ENCODE'ing each of the argument expressions for the declaration, defining internal functions in terms of those expressions, and defining the error calculation expressions (for position, orientation, etc.) in terms of those internal functions. The mechanism of servo declaration is described more fully in the PREDEF discussion below. Servo control commands merely set or clear flags which are checked by the interrupt level software.

PREDEF

System functions and servo error calculation expressions are defined in this module. The system functions fall into several classes:

1. Constant functions have their values hard-coded, and have no definition pointers. ZERO, BX, BY, and B$Z are in this class.

2. Dynamically updated functions have no definition pointers, but their values are constantly updated by the interrupt level software. R$WRIST and R$FINGER are in this class.

3. Pre-defined functions have hard coded definitions generated by the assembler. Most of these functions are internal and are used for servo error calculation.

4. Routine-defined functions have hard-coded subroutines as their definitions. To evaluate these functions, the subroutines are executed.

The pre-defined functions GPOS and GFOR are used by the interrupt level software servo to calculate the goal wrist position and goal wrist force. Each have five definitions, one of which is selected by the current position servo declaration. In the following description, the vector expressions VEXP1 and VEXP2 are tokenized and used as the definitions for the internal functions PFUN1 and PFUN2 respectively. PFUN3 is defined by a routine which calculates the direction of VEXP3. FEXP is a vector expression used as the definition for FFUN, the desired force function. The five hard-coded GPOS and GFOR definitions use these functions.

Below are the hard coded definitions for GPOS and GFOR for each position servo mode:

```
position none
     GPOS:=R$WRIST
     GFOR:=FFUN

position goal VEXP1 = VEXP2
     GPOS:=R$WRIST+PFUN2-PFUN1
     GFOR:=FFUN

position point VEXP1 = VEXP2
     GPOS:=R$WRIST+PFUN2-PFUN1
     GFOR:=-.9*R$FORCE

position line VEXP1 = VEXP2;VEXP3
     GPOS:=R$WRIST+PFUN2-PFUN1-(PFUN2-PFUN1).PFUN3*PFUN3
     GFOR:=FFUN.PFUN3*PFUN3-.9*(R$FORCE-R$FORCE.PFUN3*PFUN3)

position plane VEXP1 = VEXP2;VEXP3
     GPOS:=R$WRIST+(PFUN2-PFUN1).PFUN3*PFUN3
     GFOR:=FFUN-FFUN.PFUN3*PFUN3-.9*(R$FORCE.PFUN3*PFUN3)
```

The pre-defined function GTOR is used by the interrupt level software to calculate the goal wrist torque. Orientation calculation also takes place at interrupt level, but it cannot be expressed solely in terms of scalar and vector operations. Instead, the vector expressions in an orientation declaration are used to calculate the rows of two 3 by 3 matrices which are used by the software servo. These two matrices are called MATA and MATB. In the following discussion, VEXP1, VEXP2, VEXP3, VEXP4, and TEXP are all vector expressions used as definitions of the internal functions OFUN1, OFUN2, OFUN3, OFUN4, and TFUN. These functions are used to define the matrices MATA and MATB and the goal torque GTOR. The definitions depend upon the orientation declaration type.

```
orient none
     MATA and MATB are unused.
     GTOR:=TFUN

orient goal VEXP1;VEXP2 = VEXP3;VEXP4
     MATA:= <OFUN3>
            <OFUN4>
            <<OFUN3>#<OFUN4>>

     MATB:= <OFUN1>
            <OFUN2>
            <<OFUN1>#<OFUN2>>

     GTOR:=TFUN

orient fixed VEXP1;VEXP2 = VEXP3;VEXP4
     MATA:= <OFUN3>
            <OFUN4>
            <<OFUN3>#<OFUN4>>
     MATB:= <OFUN1>
            <OFUN2>
            <<OFUN1>#<OFUN2>>

     GTOR:=-.9*R$FORCE

orient alignVEXP1 = VEXP3
     MATA:= <OFUN3>
            <(<OFUN3>-<OFUN1>)#<OFUN1>>
            <OFUN3>#<(<OFUN3>-<OFUN1>)#<OFUN1>>

     MATB:= <OFUN1>
            <(<OFUN3>-<OFUN1>)#<OFUN1>>
            <OFUN1>#<(<OFUN3>-<OFUN1>)#<OFUN1>>

     GTOR:=TFUN.<OFUN3>*<OFUN3>-.9*(R$TORQUE-R$TORQUE.<OFUN3>*<OFUN3>)
```

The scalar expression included in hand commands is used to define the scalar function hfun. This function is used by the software servo to control hand opening or squeezing force.

F.2. INTERRUPT LEVEL SERVO CALCULATIONS

The actual servo operations are performed by a sampled-data servo which runs at interrupt level. This servo is driven by a 60 Hz. clock, and is executed every fourth clock tick. The finger squeeze servo requires a faster sampling rate and is executed every clock tick. The entire software servo is found in the module RSERVO.

Since some background tasks require multiple words to be changed without intervening clock ticks, a mechanism is provided to temporarily delay a clock tick, without losing it. The interrupt vector, which normally points to the servo routine, is changed to point to a routine which sets a flag if the clock ticks. After the critical code has been executed, the vector is reset, and if the flag is set, a fake clock interrupt is generated.

Every fourth clock tick, the main software servo routine is executed. First, all user-defined function values are flagged as invalid; therefore, the first reference to these functions will cause them to be re-evaluated. Successive references will simply return the previous value.

Next, the data link to the PDP10 is checked to see if any messages have been sent. If not, the vision processor sampling interval is increased by one. The variable TVIS contains the number of sampling intervals since the last valid sample. If the vision processor is running fast enough, TVIS will be one. The messages from the vision processor cause lost or found flags to be set in the vision function status words, and update the values of vision functions if appropriate.

Next, the robot joint positions are read and the robot wrist position R$WRIST, and the robot hand orientation vectors R$FINGER and R$THUMB are calculated. These values are stored in the value field of the corresponding functions in PREDEF.

The external joint torques are calculated by calling the routine CALCTX. This routine uses the difference between the actual joint velocity, and the velocity indicated by the hardware servo's velocity feedback to estimate the torque being applied to the joint, as described in Appendix D.

The state of the robot panic button is checked. If it was pressed, the robot is stopped and the servo calculation routines are deactivated. The values of system-defined functions are still updated.

The routine CALCGR is called to calculate the gravity compensation for each joint.

The internal system function GPOS is evaluated to find the goal wrist position. If GPOS uses a vision function, a check is made to guarantee that the feature was not lost and that the vision processor has updated its value during this sample time. If so, the new GPOS value is used as the goal wrist position. If not, the last valid value of GPOS is retained.

Next, the hand orientation vectors are found. If no orientation constraint is declared, the current orientation is used as the goal. Otherwise, the routine ORROT is called to calculate a rotation matrix which indicates how the hand should be rotated with respect to its current position. This rotation matrix, $\underline{\mathrm{ROTMAT}}$, is calculated by:

$$\underline{\mathrm{ROTMAT}} = \underline{\mathrm{MATA}} * \underline{\mathrm{MATB}}^{-1}$$

where $\underline{\mathrm{MATA}}$ and $\underline{\mathrm{MATB}}$ are determined by the orientation declaration as described previously. The two hand orientation vectors R$FINGER and R$THUMB are multiplied by this matrix to obtain the goal values for R$FINGER and R$THUMB. If the orientation depends upon a vision function, the function must be valid, otherwise the last valid orientation is retained.

Now the force and torque transformation matrices are calculated based upon the current arm position, using the method described in Appendix B.

Next, the goal values for the wrist position and hand orientation are used to find the goal joint angles. This procedure is described in section 5.3. A wrist flip flag determines if a joint limit error will cause wrist flipping, or simply cause the joint to saturate at its limit.

The external forces on the joints and the previously calculated force and torque transformation matrices are used to calculate the wrist external force vector R$FORCE and the wrist external torque vector R$TORQUE, as described in Appendix B. These values are stored in the value field of the corresponding functions in PREDEF.

Next, the goal wrist force and torque functions GFOR and GTOR are evaluated. The values obtained are transformed into joint torques using the previously calculated transformation matrices. Note that if these functions reference R$FORCE or R$TORQUE, they will receive the value calculated in the previous step.

The routine CALCSE is called for each joint to calculate the joint current based upon the actual and goal joint angles, the gravity compensation, and the desired joint torque. The overall velocity gain is scaled by TVIS to guarantee stability even if the vision processor is too slow. This gain is also scaled so that no joint current value exceeds the limit for that joint. Therefore, all joints will slow down to track the slowest one.

Finally, the routine DOSERV writes all the joint current values to the appropriate D/A registers in the hardware servo.

After the main joint servoing has been completed, the hand servo calculations are performed. The desired hand mode is checked, and if necessary, the the hand function hfun is evaluated. If a squeeze operation is specified, the squeeze value is calculated and stored for the squeeze servo.

At this point, all the computations indicated by the interrupt level blocks in Figure 13 have been completed, and the interrupt routine exits.

F.3. RSS PROGRAM EDITING AND EXECUTION

This section contains the User Command Processor, the Editor, the Program Command Processor, and the Condition Monitor Processor shown in Figure 13.

The User Command Processor consists mainly of the module COMPRO. The routine COMEX reads a single command from the user terminal or disk file, and takes the appropriate action. If the command is not a top level command, it is assumed to be a servo command and is passed on to the Servo Command Processor.

The Editor is distributed between COMPRO and PROGEX. COMPRO contains the routines which parse editing commands, and PROGEX performs the actual editing functions. The editor is very simple and is limited to line oriented insertions and deletions.

All of the programming capability of RSS was added after the servo command processor and interrupt level servo were completed. The program is stored internally as a linked list of ASCIZ strings, each of which is either a program command or a servo command. Each of these strings represents a single program line and may have a label. Program execution is handled by the routines in the module PROGEX. Each program has a status block which contains: the program name, the program status, a pointer to the first program line, a pointer to the current editing position, a pointer to the next line to execute, and a pointer to the symbol table. The routine RUNNER is called to continue execution of the currently active program. RUNNER first checks the condition monitors to see if any have fired by calling the routine DEMCHK. If the program is not done, in wait state, or stopped, the routine ROUTEX is called to execute the program statements. ROUTEX checks each statement to see if it is a program command. If so, it performs the desired actions: defining condition monitors, branching, or waiting. Otherwise it passes the command on to the Servo Command Processor. Statements are executed until wait state is entered, a pause command is executed, the program ends, or an error is detected.

The Condition Monitor Processor is distributed between PROGEX and a module called DEMON. The condition monitors are a linked list of blocks which contain: status bits, a pointer to the expression or word to monitor, and a pointer to the program statement which should get control if the condition is satisfied. The list of monitors is stored in order of decreasing priority. Each time the routine DEMCHK is called, the list is scanned for active condition monitors. For each active monitor, the routine DEMEVAL is called to evaluate the expression or to check the specified flag bits in order to see if the condition is satisfied. If so, control is passed to the indicated program statement, and the program is removed from wait state. If a condition is satisfied, all monitors of equal or lower priority are deleted. The condition monitors are only checked when the program is in wait state, in pause state, or when a branch statement is executed. Therefore, servos which require several lines to define will not be interrupted by a condition monitor.

The non-interrupt portion of RSS is merely a loop which is executed continually:

1. The next command (if any) from the user terminal or require file is executed.

2. If an RSS program is active, the condition monitors are checked to see if any have fired.

3. If an RSS program is active and not in wait or pause state, program statements are executed until wait state is entered, a pause statement is executed, a branch occurs, the program stops, or an error is detected.

4. Go to 1.

VITA

Clifford Calvin Geschke was born on February 16, 1953 in Oak Park, Illinois. He attended Maine Township High School South in Park Ridge, Illinois and graduated in 1970. He attended Purdue University in West Lafayette, Indiana, and graduated with distinction in 1973 with a Bachelor of Science in Electrical Engineering. He then enrolled in the Graduate College of the University of Illinois at Urbana-Champaign and worked as a Graduate Research Assistant in the Advanced Automation Research Group of the Coordinated Science Laboratory. After receiving his Master of Science in Electrical Engineering, he continued working at the Coordinated Science Laboratory where he completed his doctoral research.